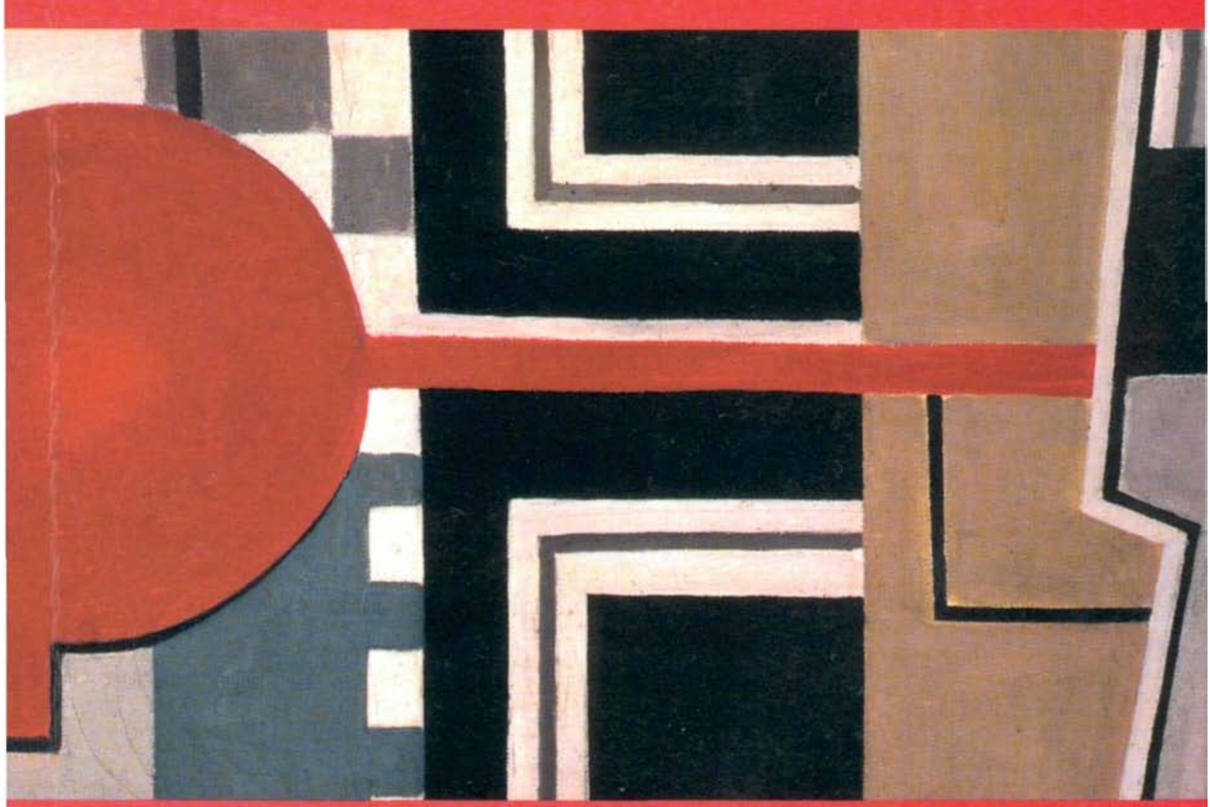
Современное проектирование на C++

Обобщенное программирование и прикладные шаблоны проектирования

Андрей Александреску

Предисловие Скотта Мейерса
Предисловие Джона Влиссидеса



Серия *C++ In-Depth* ◆ Бьярн Страуструп

Современное проектирование на C++

Обобщенное программирование и прикладные шаблоны проектирования

Андрей Александреску



Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2002

ББК 32.973.26-018.2.75
А46
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *А.В. Слепцов*

Перевод с английского и редакция канд.физ.-мат.наук *Д.А. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Александреску, Андрей.

А46 Современное проектирование на С++. Серия *С++ In-Depth*, т. 3.: Пер. с англ. — М. : Издательский дом "Вильямс", 2002. — 336 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0351-3 (рус.)

В книге изложена новая технология программирования, представляющая собой сплав обобщенного программирования, метапрограммирования шаблонов и объектно-ориентированного программирования на С++. Обобщенные компоненты, созданные автором, высоко подняли уровень абстракции, наделив язык С++ чертами языка спецификации проектирования, сохранив всю его мощь и выразительность.

В книге изложены способы реализации основных шаблонов проектирования. Разработанные компоненты воплощены в библиотеке Loki, которую можно загрузить с Web-страницы автора. Книга предназначена для опытных программистов на С++.

ББК 32.973.26-018.2.75

ISBN 5-8459-0351-3 (рус.)
ISBN 0-201-77581-6 (англ.)

Оглавление

Содержание

Предисловие Скотта Мейерса

В 1991 году вышло первое издание книги "Эффективное использование C++" ("Effective C++"). В нем почти не рассматривались шаблоны, поскольку в то время они были новшеством, и я о них практически ничего не знал. Включенные в книгу немногочисленные фрагменты программ, содержащих шаблоны, я был вынужден посылать по электронной почте другим людям, поскольку все доступные мне компиляторы их не поддерживали.

В 1995 году я написал книгу "Наиболее эффективное использование C++" ("More effective C++"). И снова почти не упомянул о шаблонах. На этот раз меня остановило не отсутствие знаний (в первоначальном варианте книга содержала целую главу, посвященную этой теме) и не ограниченность моих компиляторов. Просто возникло подозрение, что понимание роли шаблонов в среде программистов на языке C++ претерпевает настолько значительные изменения, что все мои мысли по этому поводу могут быстро стать банальными, поверхностными и даже ошибочными.

Эти подозрения возникли по двум причинам. Первой из них была статья, опубликованная в январском номере журнала "C++ Report" за 1995 год Джоном Бартоном (John Barton) и Ли Нэкманом (Lee Nackman). В ней описывалось применение шаблонов для безопасного анализа размерностей (typesafe dimension analysis) без дополнительных затрат машинного времени. Я сам довольно долго пытался решить эту задачу и знал, что другие программисты также безуспешно ломают над ней голову. Революционный подход, предложенный Бартоном и Нэкменом, помог мне понять, что с помощью шаблонов можно не просто создавать контейнеры, содержащие объекты класса T, но и достичь намного более значительных результатов.

В качестве иллюстрации этого подхода рассмотрим код, предназначенный для умножения двух физических величин произвольной размерности.

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                        Physical<m2, l2, t2> rhs)
{
   return  Physical<m1+m2,l1+l2,t1+t2>::
           unit*lhs.value()*rhs.value();
}
```

Даже без объяснений, приведенных в статье, совершенно очевидно, что эта шаблонная функция (function template) получает шесть параметров, ни один из которых не представляет собой какой-либо тип! Это явилось для меня приятным открытием.

Вскоре я стал изучать стандартную библиотеку шаблонов STL (Standard Templates Library). Эта разработка Александра Степанова (Alexander Stepanov) весьма элегантна. В ней контейнеры ничего не знают об алгоритмах, алгоритмы ничего не знают о контейнерах, итераторы функционируют как указатели (но могут быть объектами), контейнеры и алгоритмы одинаково успешно могут получать указатели на функции и сами функции в виде объектов, а пользователи библиотеки могут расширять ее, не прибегая к наследованию от какого-либо базового класса или переопределению виртуальных функций. И тут я почувствовал (как и при чтении статьи Бартона и Нэкмена), что практически *ничего* не знаю о шаблонах.

По этой причине я не стал почти ничего писать о шаблонах в книге "Более эффективный C++". Как я мог касаться этой темы, если мое понимание шаблонов ос-

тавалось на уровне контейнеров, содержащих объекты класса T, в то время как Бартон, Нэкман, Степанов и другие продемонстрировали, что такое применение шаблонов является лишь вершиной айсберга?

В 1998 году Андрей Александреску (Andrei Alexandrescu) и я стали обмениваться сообщениями по электронной почте, и вскоре я понял, что мне придется снова изменить свое мнение о шаблонах. В то время как Бартон, Нэкман и Степанов ошеломили меня тем, *что* можно сделать с помощью шаблонов, Андрей поразил меня, объяснив, *как* это можно сделать.

Одна из простейших вещей, которые он помог мне изложить в общедоступной форме, до сих пор остается примером, который я первым привожу людям, начинающим работать в этой области. Это шаблон CTAssert, представляющий собой аналог макроса assert, но позволяющий проверять условия во время компиляции, а не во время выполнения программы.

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

Вот и все! Обратите внимание на то, что обычный шаблон CTAssert нигде не определяется. Более того, он конкретизирован только для значения true, но не для false. То, что есть в этом шаблоне, не менее важно, чем то, *чего в нем нет*. Это заставляет посмотреть на код этого шаблона под другим углом, поскольку оказывается, что большая часть его "исходного кода" осознанно проигнорирована. Этот образ мышления совершенно отличается от общепринятого. (В этой книге Андрей обсуждает более сложный шаблон CompileTimeCheker.)

В итоге Андрей приступил к разработке шаблонно-ориентированной реализации распространенных языковых идиом (language idioms) и шаблонов проектирования, особенно шаблонов GoF[1]. Это вызвало перепалку в среде разработчиков шаблонов, поскольку они были абсолютно убеждены, что эти шаблоны невозможно запрограммировать.

Когда стало ясно, что Андрей создал средства для автоматического генерирования *реализаций* шаблонов, а не для программирования собственно шаблонов, возражения были сняты. Мне было приятно узнать, что Андрей и один из разработчиков шаблонов GoF (Джон Влиссидес) вместе написали две статьи в журнале "C++ Report", посвященные этой теме.

Следуя выбранному направлению, связанному с шаблонами для генерации идиом и реализациями шаблонов проектирования, Андрей столкнулся с проблемами, стоящими и перед другими программистами, работающими в этой области. Должна ли программа быть безопасной в многопоточной системе (thread safe)? Откуда брать дополнительную память: из кучи, стека или пула статической памяти? Нужно ли перед разыменованием интеллектуальных указателей (smart pointers) проверять, равны ли они нулю? Что случится при завершении программы, если один деструктор синглтона (singlton's destructor) попытается использовать уже уничтоженный синглтон? Андрей стремился предложить пользователям максимально широкий выбор возможностей, не навязывая своего мнения.

Он решил инкапсулировать эти решения в виде *классов стратегий* (policy classes), что позволило пользователям передавать их как шаблонные параметры. Кроме того, Андрей предложил для этих классов разумные значения по умолчанию, так что боль-

[1] Название этих шаблонов происходит от словосочетания "Gang of Four" — "Банда четырех". В состав этой "банды" входили Erich Gamma (Эрих Гамма), Richard Helm (Ричард Хелм), Ralph Johnson (Ральф Джонсон) и John Vlissides (Джон Влиссидес), написавшие основополагающую книгу *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

щинство клиентов может их просто игнорировать. Результат оказался потрясающим! Например, шаблон для интеллектуального указателя, описанный в книге, получает в виде параметров только 4 класса стратегий, а генерирует более 300 разных типов интеллектуальных указателей, каждый из которых обладает уникальными особенностями поведения. Программисты, осведомленные о поведении интеллектуального указателя, заданном по умолчанию, могут вообще проигнорировать параметры, представляющие собой классы стратегий, указав лишь тип объекта, на который он должен ссылаться. Это позволяет им извлекать выгоду из прекрасно сделанного класса для интеллектуального указателя, не прилагая никаких усилий.

В конце книги рассмотрены три разные темы, причем для каждой из них выбран свой способ изложения. Во-первых, представлен новый взгляд на мощь и гибкость шаблонов в языке C++. (Если, прочитав материал, посвященный спискам типов (typelists), вы не свалились со стула, значит, вы сидели на полу.) Во-вторых, указаны ортогональные направления, по которым идиомы и реализации шаблонов могут отличаться друг от друга. Для разработчиков шаблонов и программистов, занимающихся их реализацией, эта информация крайне важна, однако вы вряд ли найдете ее в других источниках. В заключение, читатели могут свободно загрузить исходный код библиотеки шаблонов Loki, описанной в этой книге, и изучить ее строение. С ее помощью вы можете не только испытать свой компилятор, но и начать собственную разработку. Разумеется, вы можете вполне законно использовать код, написанный Андреем, для своих целей. Я абсолютно уверен, что ему это будет приятно.

Скотт Мейерс (Scott Meyers)

Предисловие Джона Влиссидеса

Что нового можно сказать о языке С++? Оказывается, очень много. Эта книга посвящена слиянию разных способов программирования — обобщенного программирования, метапрограммирования шаблонов, объектно-ориентированного программирования и разработки шаблонов проектирования — в рамках нового подхода. До сих пор эти направления в программировании развивались изолированно друг от друга, и выгоды, полученные от их объединения, лишь начинают получать достойную оценку. Это слияние открывает новые перспективы для языка С++ не только с точки зрения собственно программирования, но для разработки программного обеспечения в целом. Особенно значительно это повлияет на анализ программного обеспечения и его архитектуру.

Обобщенные компоненты, созданные Андреем, поднимают уровень абстракции настолько высоко, что язык С++ приобретает черты языка спецификаций проектирования (design specification language). При этом в отличие от узкоспециализированных языков проектирования язык С++ сохраняет всю свою мощь и выразительность. Андрей продемонстрировал, как программируются концепции проектирования: синглтоны (singletones), инспекторы (visitors), заместители (proxies), абстрактные фабрики (abstract factories) и т.п. Можно даже настраивать готовые компоненты с помощью шаблонных параметров, не расходуя дополнительного машинного времени. Не нужно выбрасывать кучу денег на разработку новых инструментальных средств или изучать тома методологической тарабарщины. Достаточно иметь надежный современный компилятор (и эту книгу).

Разработчики генераторов кода долгие годы обещали обеспечить их совместимость, но теоретические исследования и практический опыт убедили меня, что достичь этой цели невозможно. Остаются нерешенными проблемы полного обхода дерева поиска, генерации недостаточно качественного кода, негибких генераторов, нечитабельности сгенерированного кода и, разумеется, широко известная проблема, которую можно сформулировать так: "Я не могу вставить этот проклятый код в свою программу". Каждой из этой проблем достаточно, чтобы завести программиста в тупик, а вместе они создают практически непреодолимые препятствия для автоматической генерации кода.

Как было бы хорошо получить все теоретические преимущества автоматической генерации кода — скорость, легкость реализации, сокращенную избыточность, меньшее количество ошибок, одновременно избежав его практических недостатков! Именно это обещает подход, предложенный Андреем. Обобщенные компоненты (generic components) реализуют удачные схемы в виде удобных для использования, поддающихся смешиванию и подходящих для решения задачи шаблонов (mixable-and-matchable templates). Эти шаблоны делают практически то же, что и генераторы кода: создают стандартные фрагменты кода для дальнейшей обработки с помощью компилятора. Отличие заключается в том, что шаблоны позволяют сделать это, не выходя за рамки языка С++. В результате происходит полная интеграция полученного кода с исходным кодом приложения. При этом остается возможность использовать всю мощь языка, расширяя классы, замещая методы и подгоняя шаблоны под свои требования.

Некоторые из описанных приемов программирования довольно трудно понять, особенно метапрограммирование шаблонов, рассмотренное в главе 3. Однако, освоив

его, вы сможете постичь всю теорию обобщенных компонентов, которые практически сами себя создают. Эти компоненты описаны в последующих главах. Я думаю, что метапрограммирование шаблонов, изложенное в главе 3, само по себе достойно отдельной книги. Остальные десять глав освещают способы его применения. Несмотря на то что десять глав — это довольно много, ваши инвестиции окупятся сторицей.

Джон Влиссидес (John Vlissides)

Предисловие

Возможно, вы держите эту книгу в руках, находясь в книжной лавке, раздумывая, покупать ли ее? А может быть вы пришли в библиотеку и решаете, стоит ли тратить время на эту книгу? Знаю, что у вас нет времени, поэтому буду краток. Если вы когда-либо интересовались, как нужно писать программы высокого уровня на языке C++, как справиться с лавиной мелких деталей, загромождающих даже самую простую программу, или как создать компонент, пригодный для повторного использования, который не нужно разрабатывать заново для каждого нового приложения, то эта книга для вас.

Представьте себе такую картину. Вы приходите с производственного совещания, неся в руках груду диаграмм, на которых нацарапаны ваши комментарии. О'кей, говорите вы, тип события, передаваемого от одного объекта к другому, в любом случае не `char`. Это — тип `int`. И вы изменяете одну строку в вашей программе. Интеллектуальный указатель на объект класса `Widget` работает слишком медленно, его следует сделать неконтролируемым. И вы изменяете еще одну строку. Фабрика объектов должна поддерживать новый класс `Gadget`, добавленный соседним отделом. И вы снова изменяете одну строку.

Вы закончили разработку своей программы. Компилируете. Связываете. Готово.

Отлично! Не кажется ли вам, что в этом сценарии что-то не так? Намного правдоподобнее выглядит следующее развитие событий. Вы приходите с производственного совещания взмыленный, поскольку вам предстоит выполнить кучу работы. Вы запускаете глобальный поиск. Удаляете фрагмент. Добавляете фрагмент. Делаете ошибки. Исправляете ошибки... В этом и заключается работа программиста, не так ли? Хотя эта книга и не гарантирует вам исполнение первого сценария, она поможет вам пройти несколько шагов в этом направлении. Здесь предпринята попытка представить язык C++ в новом качестве — языка для разработки архитектуры программного обеспечения.

Традиционно код представляет собой наиболее детализированный и сложный аспект программного обеспечения. Исторически, несмотря на существование языков разных уровней, предназначенных для поддержки методологий проектирования (например, объектной ориентации), между проектом программы и ее кодом лежит пропасть. Это обусловлено тем, что в коде должны быть учтены все мельчайшие детали реализации и множество других побочных моментов. Цель программы в большинстве случаев скрывается за множеством подробностей.

В этой книге представлена коллекция пригодных к повторному использованию шаблонов, называемых ***обобщенными компонентами*** (generic components), а также способы их разработки. Обобщенные компоненты предоставляют пользователю хорошо известные выгоды, свойственные библиотекам, однако они пригодны для более широкого спектра системных архитектур. Приемы кодирования и реализации сконцентрированы на задачах и моментах, традиционно присущих проектированию, которое обычно ***предшествует*** собственно кодированию программ. Благодаря своему высокому уровню абстракции обобщенные компоненты позволяют необычайно выразительно, сжато и легко отображать в коде сложные архитектуры.

В обобщенных компонентах воплощены три ветви программирования: проектирование шаблонов, обобщенное программирование и язык C++. Комбинация этих элементов позволила достичь высокого уровня готовности кода к повторному использо-

ванию, условно говоря, как в горизонтальном, так и в вертикальном направлениях. В горизонтальном направлении небольшое количество библиотечного кода позволяет реализовать огромное — в принципе, бесконечное — количество структур и моделей поведения. В вертикальном направлении степень обобщенности этих компонентов делает их пригодными для широчайшего круга программ.

Своим появлением книга обязана проектированию шаблонов, которое позволяет создавать мощные средства решения часто встречающихся задач объектно-ориентированной разработки программ. Проектирование шаблонов — это тщательно отобранные примеры хорошей разработки — рецепты правильных, пригодных к повторному использованию решений задач, возникающих в различных областях. Основной задачей проектирования шаблонов является создание содержательного лексикона (suggestive lexicon) для воплощаемых разработок. Эти шаблоны описывают задачу, ее проверенное временем решение с разными вариантами, а также последствия выбора одного из них. Проектирование шаблонов не связано с конкретными языками программирования. Следуя определенным шаблонам проектирования и комбинируя их друг с другом, компоненты, представленные в этой книге, стремятся охватить как можно более широкий круг конкретных задач.

Обобщенное программирование — это парадигма, в центре которой лежат абстрактные типы, узкий набор функциональных требований и алгоритмы реализации, выраженные в терминах этих требований. Поскольку алгоритмы сами определяют точную и тесную связь с типами, которыми они могут манипулировать, один и тот же алгоритм можно применять для работы с широким спектром типов. Для реализации алгоритмов, приведенных в этой книге, использованы методы обобщенного программирования, позволяющие достичь минимальной специфичности, невероятной лаконичности и эффективности, присущих программам, тщательно разработанным вручную.

В качестве средства реализации в книге используется только язык C++. В этой книге вы не найдете кода, реализующего изящные системы оконного интерфейса, сложных библиотек для сетевого программирования или интеллектуальных механизмов регистрации (logging mechanisms). Вместо этого вы обнаружите массу базовых компонентов, которые облегчают решение как всех описанных выше задач, так и многих других. Для разработки этих компонентов язык C++ крайне необходим. Лежащий в его основе механизм управления памятью, реализованный в языке C, гарантирует быстрое выполнение программ, поддержка полиморфизма позволяет применять приемы объектно-ориентированного программирования, а шаблоны допускают использование невероятных машин, предназначенных для автоматической генерации кода. Шаблоны проходят красной нитью через всю книгу, поскольку они обеспечивают тесное взаимодействие пользователя и библиотеки. Пользователь библиотеки буквально контролирует способ генерации кода, причем этот способ ограничивается самой библиотекой. Предназначение библиотеки обобщенных компонентов — позволять пользователю создавать собственные типы и определять их поведение, а также правильно объединять их с другими обобщенными компонентами. Поскольку при этом используются статические методы, ошибки, связанные со смешиванием и подгонкой соответствующих фрагментов, обычно обнаруживаются на этапе компиляции.

Аудитория

Аудитория, которой предназначена эта книга, состоит из двух частей. К первой категории относятся опытные программисты на C++, желающие овладеть наиболее современными методами создания библиотек. В книге представлены новые мощные

идиомы языка C++, обладающие удивительными возможностями, некоторые из которых невозможно было себе представить. Эти идиомы окажут неоценимую помощь при создании библиотек высокого уровня. Для программистов среднего уровня, желающих повысить свою квалификацию, книга также будет полезной, особенно если они проявят определенную настойчивость. Несмотря на то что иногда в книге встречаются довольно сложные фрагменты кода на C++, они всегда сопровождаются подробным комментарием.

Вторая категория состоит из постоянно занятых программистов, которым нужно сделать дело, не тратя лишнего времени на изучение теории. Они могут пропустить наиболее сложные детали реализации и сосредоточить свое внимание на *использовании* предложенной библиотеки. Каждая глава начинается с подробного введения и заканчивается разделом, посвященным часто задаваемым вопросам. Для понимания и использования компонентов эти разделы окажутся весьма полезными. Компоненты можно изучать независимо друг от друга. Они очень мощны и тем не менее безопасны, и, кроме того, их очень легко применять в своих приложениях.

От читателя требуется хорошее знание языка C++ и желание знать его еще лучше. Следует также иметь представление о шаблонах вообще и стандартной библиотеке шаблонов (STL) в частности.

Знание основных шаблонов проектирования (Гамма и др., 1995) желательно, но не обязательно. Идиомы и шаблоны, применяемые в книге, детально описаны. Однако эта книга посвящена другой теме — в ней не делается попыток максимально обобщить шаблоны проектирования. Поскольку они рассматриваются с точки зрения прагматичного создателя библиотеки, даже читатель, интересующийся в основном шаблонами проектирования, найдет для себя много нового, если захочет.

Библиотека Loki

В книге описывается реальная библиотека Loki, написанная на языке C++. Локи (Loki) — это бог остроумия в скандинавской мифологии, и автор надеется, что оригинальность и гибкость этой библиотеки соответствует названию. Все элементы библиотеки находятся в пространстве имен `Loki`. В примерах программ пространство имен не указывается, поскольку это увеличило бы размер кода и затемнило его содержание. Библиотеку Loki можно свободно загрузить с Web-страницы `http://www.awl.com/cseng/titles/0-201-70431-5`.

За исключением части, касающейся потоков, библиотека Loki написана на стандартном языке C++. Увы, это означает, что многие современные компиляторы не смогут работать с ней в полном объеме. Я реализовал и протестировал библиотеку Loki с помощью компиляторов CodeWarrior Pro 6.0 компании Metrowerks и Comeau C++ 4.2.38 (оба компилятора работали под управлением системы Windows). Похоже, что компилятор KAI C++ также не должен иметь с этой библиотекой никаких проблем. Как только поставщики распространят новые, усовершенствованные версии компиляторов, вы сможете эксплуатировать библиотеку Loki полностью.

Код библиотеки Loki, а также примеры, приведенные в книге, используют популярный стандарт кодирования, предложенный Хербом Саттером (Herb Sutter). Я уверен, что вы легко его поймете. Этот стандарт сводится к следующему.

- Классы, функции и перечислимые типы выглядят так: `LikeThis`.
- Переменные и перечислимые значения выглядят так: `likeThis`.
- Переменные-члены выглядят так: `likeThis_`.

- Шаблонные параметры объявляются с ключевым словом `class`, если они представляют собой тип, определяемый пользователем (user-defined type), и с ключевым словом `typename`, если тип является простым (primitive).

Структура книги

Книга состоит из двух основных частей: способы программирования и компоненты. Часть I (главы 1–4) описывает способы программирования на языке C++, используемые в обобщенном программировании и, в частности, для создания обобщенных компонентов. Представлено множество особенностей и способов программирования на языке C++: проектирование, основанное на анализе поведения, частичная специализация шаблонов, списки типов, локальные классы и т.д. Эту часть можно читать последовательно, а затем возвращаться к ней за конкретной информацией.

Часть II организована так же, как и часть I. В ней рассматривается реализация обобщенных компонентов. Здесь нет искусственных примеров. Все описанные компоненты используются в реальных приложениях. Проблемы, ежедневно встающие перед программистами на языке C++, например, интеллектуальные указатели, фабрики объектов и функторы, обсуждаются глубоко и решаются в общем виде. Реализации, приведенные в тексте, ориентируются на основные потребности программистов и предназначены для решения фундаментальных задач. Вместо подробного объяснения, что именно делает тот или иной фрагмент кода, в книге последовательно применяется следующий подход: сначала обсуждается задача, а затем выбирается и реализуется метод ее решения.

Глава 1 посвящена классам стратегий — идиомам языка C++, позволяющим создавать гибкие шаблоны.

В главе 2 обсуждаются основные способы программирования на языке C++, относящиеся к обобщенному программированию.

Списки типов, представляющие собой мощные структуры для манипуляции с типами, реализуются в главе 3.

В главе 4 описывается важный вспомогательный инструмент — механизм распределения памяти для небольших объектов (small-object allocator).

Обобщенные функторы, использующие шаблон проектирования `Command`, обсуждаются в главе 5.

В главе 6 описываются синглтоны.

Глава 7 посвящена интеллектуальным указателям.

В главе 8 описываются обобщенные фабрики объектов.

Глава 9 посвящена шаблону проектирования `Abstract Factory` и его реализации.

В главе 10 в общем виде реализовано несколько вариантов шаблона проектирования `Visitor`.

Механизмы мультиметодов (multimethod engines), представляющие собой решения, ориентированные на использование готовых компонентов, реализованы в главе 11.

Благодарности

Прежде всего хочу поблагодарить моих родителей за их постоянную заботу.

Следует подчеркнуть, что этой книги, как и большинства моих профессиональных успехов, не было бы без Скотта Мейерса. С момента нашей встречи на Всемирном конгрессе по C++ (C++ Worlds Congress) в 1998 году Скотт постоянно помогал мне. Он первым с энтузиазмом поддержал мои ранние идеи. Скотт познакомил меня с Джоном Влиссидесом, положив начало нашему сотрудничеству. Он посоветовал Хербу Саттеру сделать меня обозревателем журнала "C++ Report" и привел в издательство Addison-Wesley, практически вынудив начать эту книгу. В конце концов Скотт своими советами и замечаниями помогал мне все время в процессе работы над книгой, разделяя со мной творческие муки.

Выражаю глубокую признательность Джону Влиссидесу, который своими резкими замечаниями убедил меня, что мои решения не идеальны, и помог их улучшить. Глава 9 — его заслуга. Она появилась в книге благодаря постоянным требованиям Джона не останавливаться на достигнутом и искать более удачные решения.

Благодарю П. Дж. Плагера (P. J. Plaeger) и Марка Брианда (Mark Briand), вдохновивших меня писать статьи в журнал "C/C++ Users Journal" в то время, когда я считал обозревателей этого журнала инопланетянами.

Я очень признателен моему редактору Дебби Лафферти (Debbie Lafferty) за ее постоянную поддержку и полезные советы.

Мои коллеги по компании RealNetworks, особенно Борис Джеркуница (Boris Jerkunica) и Джим Кнаак (Jim Knaak), очень помогли мне, создав атмосферу свободомыслия, соперничества и стремления к вершинам мастерства. Я очень благодарен им за это.

Выражаю свою признательность всем участникам форумов comp.lang.c++.moderated и comp.std.c++.Usenet. Эти люди помогли мне лучше понять язык C++.

Я хотел бы выразить свою благодарность рецензентам моей рукописи: Михаилу Антонеску (Mihail Antonecku), Бобу Арчеру (Bob Archer) (моему самому строгому рецензенту), Аллену Бродману (Allen Broadman), Ионату Бурете (Ionut Burete), Мирель Чирита (Mirel Chirita), Стиву Кламагу (Steve Clamage), Джеймсу Коплину (James Coplien), Дугу Хазену (Doug Hazen), Келвину Хенни (Kelvin Henney), Джону Хикину (John Hickin), Говарду Хиннанту (Howard Hinnant), Сорину Джиану (Sorin Jianu), Золтану Кормошу (Zoltan Kormos), Джеймсу Кайперу (James Kuyper), Лизе Липпинкот (Lisa Lippincott), Джонатану Лундквисту (Jonathan Lundquist), Петру Маргиняну (Petru Marginean), Патрику МакКиллену (Patrick McKillen), Флорину Михайлу (Florin Mihaila), Сорину Опря (Sorin Oprea), Джону Поттеру (John Potter), Адриану Рапитяну (Adrian Rapiteanu), Монике Рапитяну (Monica Rapiteanu), Брайану Стентону (Brian Stenton), Адриану Стефле (Adrian Steflea), Хербу Саттеру (Herb Sutter), Джону Торйо (John Torjo), Флорину Трофину (Florin Trofin) и Кристи Власяну (Cristi Vlaseanu). Все они внесли свой вклад в улучшение рукописи. Без них мне не удалось бы сделать и половины работы.

Спасибо Грегу Комо (Greg Comeau) за то, что он предоставил в мое распоряжение превосходный компилятор.

В заключение я хотел бы поблагодарить всю мою семью и друзей за их постоянное поощрение и поддержку.

Часть I

Методы

1

Разработка классов на основе стратегий

В этой главе описываются понятия стратегии (policy) и классов стратегий (policy classes), играющие важную роль в создании библиотек, эффективно обеспечивающих повторное использование кода. Библиотека Loki была задумана именно такой. Кратко говоря, проектирование, основанное на *стратегиях*, позволяет создавать класс со сложным поведением из множества маленьких классов (называемых *стратегиями*), каждый из которых отвечает только за один функциональный или структурный аспект. Как следует из ее названия, стратегия устанавливает интерфейс, соответствующий определенному свойству. Стратегии можно реализовывать по-разному, поскольку их интерфейсы целиком находятся в компетенции программиста.

Смешивая и подгоняя стратегии друг к другу, можно моделировать огромное количество видов поведения, используя небольшой набор элементарных компонентов.

Стратегии используются во многих главах этой книги. Обобщенный шаблонный класс `SingletonHolder` (глава 6) с помощью стратегий управляет длительностью функционирования и обеспечивает безопасность работы в многопоточной среде. Класс `SmartPtr` (глава 7) почти целиком сконструирован из стратегий. Механизм двойной диспетчеризации (double-dispatch engine), описанный в главе 11, использует стратегии для выбора разных компромиссов. Реализация обобщенной "абстрактной фабрики" (Gamma et al., 1995), приведенная в главе 9, применяет стратегию для выбора метода создания объектов.

В этой главе описывается предназначение стратегий, рассматриваются детали их проектирования и приводятся советы, позволяющие разложить класс на стратегии.

1.1. Разнообразие методов разработки программного обеспечения

Область проектирования программного обеспечения как никакая другая техническая дисциплина демонстрирует большое разнообразие методов: одну и ту же задачу можно правильно решить самыми разными способами, причем между правильным и неправильным решением лежит бесконечное количество нюансов. Каждый новый способ открывает новый мир. При выборе одного из решений возникает множество возможных вариантов, начиная с уровня системной архитектуры и заканчивая малейшими деталями кодирования. Таким образом, разработка программных систем заключается в выборе решений из гигантского числа вариантов.

Рассмотрим простейший пример низкоуровневой разработки — интеллектуальные указатели (глава 7). Интеллектуальные указатели (smart pointers) представляют собой классы, которые могут быть как одно- так и многопоточными. Они могут использовать различные стратегии владения ресурсами, а также разные компромиссы между безопасностью и скоростью. Кроме того, интеллектуальные указатели могут поддерживать или не поддерживать автоматическое преобразование в обычные указатели. Все эти свойства можно свободно комбинировать, причем обычно в конкретной области, для которой предназначено программное обеспечение, наилучшим является лишь одно решение.

Разнообразие методов создания программного обеспечения постоянно смущает начинающих разработчиков. Перед ними стоит конкретная задача. Что применить для ее успешного решения? События? Объекты? Наблюдатели? Обратные вызовы? Виртуальные классы? Шаблоны? Если абстрагироваться от конкретных деталей, разные решения окажутся эквивалентными.

В отличие от новичка, опытный разработчик программного обеспечения знает, *что работает, а что — нет.* Для каждой конкретной задачи существует множество методов решения. Они обладают своими преимуществами и недостатками, которые определяют, насколько тот или иной метод подходит для решения поставленной задачи. Решение, которое казалось вполне приемлемым на бумаге, на практике может оказаться ошибкой.

Создавать программное обеспечение сложно, поскольку разработчик постоянно должен делать *выбор.* А в программировании, как и в жизни вообще, трудно сделать правильный выбор.

Опытный разработчик знает, какой выбор ведет к поставленной цели, в то время как новичок каждый раз делает шаг в неизвестность. Опытный архитектор программного обеспечения напоминает хорошего шахматиста: он видит на много ходов вперед. Для этого нужно долго учиться. Возможно поэтому гениальные программисты бывают очень молодыми, в то время как гениальные разработчики программного обеспечения обычно находятся в зрелом возрасте.

Кроме головоломок, постоянно возникающих перед начинающими разработчиками, комбинаторная природа архитектурных решений доставляет много хлопот создателям библиотек. Для того чтобы реализовать библиотеку, пригодную для создания программного обеспечения, разработчик должен классифицировать и адаптировать множество типичных ситуаций. Библиотеку следует оставить открытой для модификаций, чтобы прикладной программист мог приспособить ее для своих нужд, создав конкретную программу.

Действительно, как упаковать гибкие, основательно разработанные компоненты в библиотеку? Как предоставить пользователю возможность настраивать эти компоненты? Как преодолеть "проклятие выбора" с помощью кода, имеющего разумные размеры? Вот каким вопросам посвящена эта глава, да и вся книга в целом.

1.2. Недостатки универсального интерфейса

Реализовать все под оболочкой универсального интерфейса — неудачное решение. Это объясняется следующими причинами.

К основным негативным последствиям такого выбора относятся интеллектуальные издержки, огромный размер и неэффективность библиотеки. Гигантские классы очень непродуктивны, поскольку на их изучение нужно тратить большие усилия, они слиш-

ком велики, а программы, использующие такие классы, работают намного медленнее, чем аналогичные программы, разработанные вручную.

Однако едва ли не самой важной проблемой, связанной с использованием универсального интерфейса, является *потеря безопасности статических типов* (static type safety). Одна из основных целей архитектуры любого программного обеспечения — воплощение некоторых аксиом "по определению". Например, нельзя одновременно создавать два объекта класса `Singleton` (глава 6) или объекты непересекающихся семейств (disjoint families) (глава 9). В идеале разработчик должен накладывать большинство ограничений еще на этапе компиляции.

В большом всеобъемлющем интерфейсе очень трудно учесть все подобные ограничения. Обычно конкретные ограничения семантически соответствуют лишь части интерфейса. Это ведет к увеличению разрыва между *синтаксической* и *семантической правильностью* программ, использующих данную библиотеку.

Рассмотрим, например, аспект реализации объекта класса `Singleton`, связанный с безопасностью работы в многопоточной среде. Если библиотека полностью инкапсулирует потоковые свойства, то пользователь конкретной, непереносимой системы не может использовать библиотеку синглтонов. Если эта библиотека предоставляет доступ к основным незащищенным функциям, возникает опасность, что программист разрушит библиотеку, написав код, который будет синтаксически правильным, но семантически неверным.

А что, если библиотека будет реализовывать разные проектные решения в виде разных классов более скромного размера? Каждый класс мог бы воплощать конкретное проектное решение. При использовании интеллектуальных указателей, например, естественно ожидать появления многочисленных реализаций: `SingletonThreadedSmartPtr`, `MultiThreadedSmartPtr`, `RefCountedSmartPtr`, `RefLinkedSmartPtr` и т.п.

Для этого подхода характерен резкий рост количества разных вариантов проектных решений. Четыре упомянутых класса могут привести к появлению новых комбинаций, например, в виде класса `SingleThreadedRefCountedSmartPtr`. Преобразование типов приведет к еще большему количеству комбинаций, которые в конце концов выйдут за рамки возможностей как программиста, так и пользователя библиотеки. Очевидно, что идти этим путем не следует. Преодолеть экспоненциальный рост вариантов с помощью грубой силы невозможно.

Такого рода библиотеки не только требуют для своей разработки и применения огромных умственных усилий, но и являются крайне негибкими. Малейшая непредвиденная настройка — например, попытка инициализировать конкретным значением интеллектуальные указатели, созданные по умолчанию, — приводит все тщательно разработанные классы библиотеки в плачевное состояние.

В процессе разработки программ на них накладываются определенные ограничения. Следовательно, библиотеки, ориентированные на разработку программ, должны давать пользователю возможность формулировать *свои собственные* ограничения, а не навязывать *встроенные*. Раз и навсегда законсервированные проектные решения могут оказаться так же непригодными для библиотек, ориентированных на разработку программ, как, например, константы, явно заданные в обычных программах. Разумеется, наборы "наиболее популярных" или "рекомендуемых" готовых решений вполне допустимы, если программист может изменять их по мере надобности.

Эти проблемы иллюстрируют неблагоприятную ситуацию, сложившуюся в области разработки библиотек. Большого количества низкоуровневых универсальных и специализированных библиотек, облегчающих разработку программ, т.е. структур высо-

кого уровня, практически нет. Это парадоксальная ситуация, поскольку любое нетривиальное приложение воплощает некие проектные решения. Следовательно, библиотеки, ориентированные на создание программ, должны были бы найти применение в большинстве приложений.

Этот пробел попытались заполнить специальные среды разработки (frameworks). Однако они в основном предназначены для того, чтобы связать приложение с конкретным проектным решением, а вовсе не для того, чтобы помочь пользователю *выбрать* и *настроить* свой проект. Если программист стремится реализовать свой оригинальный проект, он должен начинать все "с нуля", создавая классы, функции и т.п.

1.3. Опасно ли множественное наследование?

Рассмотрим класс `TemporarySecretary`, производный от классов `Secretary` и `Temporary` одновременно[1]. Класс `TemporarySecretary` обладает свойствами обоих классов, описывая как секретаря, так и временно нанятого служащего. Кроме того, он может иметь и свои особенности. Это наталкивает на мысль, что множественное наследование может помочь нам справиться с экспоненциальным ростом количества проектных решений с помощью малого числа тщательно подобранных базовых классов. В таком случае пользователь смог бы создать класс для многопоточного интеллектуального указателя, подсчитывающего количество ссылок (reference-counted smart ponter), выведя его из некоторого класса `BaseSmartPtr` и классов `MultiThreaded` и `RefCounted`. Любой опытный разработчик классов знает, что этот наивный подход не работает.

Анализ причин, по которым множественное наследование не позволяет воплощать гибкие проектные решения, помогает найти правильный выход. Проблемы заключаются в следующем.

1. *Механика.* Не существует стандартного кода, позволяющего объединить наследуемые компоненты в одно целое стандартным образом. Единственным инструментом, дающим возможность объединить классы `BaseSmartPtr`, `MultiThreaded` и `RefCounted`, является языковый механизм, называемый *множественным наследованием* (multiple inheritance). Для объединения базовых классов применяется простая суперпозиция и устанавливаются правила доступа к их членам. Это совершенно недопустимо и работает лишь в простейших случаях. Чаще всего для достижения желаемого эффекта программист вынужден тщательно настраивать поведение наследуемых классов.

2. *Информация о типах.* Базовые классы не имеют достаточно информации о типах, чтобы выполнить свою работу. Представим, например, что мы пытаемся реализовать глубокое копирование интеллектуального показателя с помощью наследования от базового класса `DeepCopy`. Какой интерфейс должен иметь класс `DeepCopy`? Очевидно, что он вынужден создавать объекты, имеющие тип, о котором ему ничего не известно.

3. *Изменение состояния.* Различные аспекты поведения, реализованные в базовых классах, должны влиять на одно и то же состояние. Это означает, что они долж-

---

[1] Этот пример создан на основе старого аргумента, который Бьярн Страуструп (Bjarn Stroustrup) привел в пользу множественного наследования в первом издании книги "The C++ Programming Language" (см. Страуструп Б. "Язык программирования C++". — М.: Мир, 1990. — *Прим. ред.*). В то время множественное наследование в языке C++ еще не было реализовано.

ны применять виртуальное наследование от базового класса, имеющего заданное состояние. Это усложняет разработку и делает ее негибкой, поскольку предполагается, что классы пользователя наследуют свойства от библиотечных классов, а не наоборот.

Комбинаторное по своей природе, множественное наследование само по себе нельзя применять для поддержки многовариантных проектных решений.

1.4. Преимущества шаблонов

Для имитации комбинаторного поведения хорошо подходят шаблоны, поскольку они позволяют генерировать код во время компиляции, основываясь на типах, указанных пользователем.

По способу настройки шаблонные классы принципиально отличаются от обычных. Если нужно реализовать какой-то специальный вариант, можно специализировать (specialize) любые функции-члены шаблонного класса, конкретизировав его. Например, если задан шаблон `SmartPtr<T>`, можно специализировать любую функцию-член класса `SmartPtr<Widget>`. Это позволяет хорошо детализировать поведение настраиваемого шаблона.

Более того, для шаблонных классов с несколькими параметрами можно использовать частичную шаблонную специализацию (как мы увидим в главе 2). Это дает возможность специализировать только некоторые из аргументов шаблонного класса. Например, с помощью специализации

```
template <class T, class U> class SmartPtr { ... };
```

можно специализировать шаблон `SmartPtr<T, U>` для класса `Widget`, а также для любого другого типа, используя следующую синтаксическую конструкцию:

```
template <class U> class SmartPtr<Widget, U> { ... };
```

Статическая и комбинаторная природа шаблонов делает их очень привлекательными для создания фрагментов проектных решений. Однако на пути реализации такого подхода существует несколько неочевидных препятствий.

1. *Невозможно специализировать структуру.* Используя лишь шаблоны, невозможно специализировать структуру класса (т.е. его данные-члены). Специализировать можно только функции.

2. *Специализация функций-членов не масштабируется.* Можно специализировать любую функцию-член шаблонного класса с одним шаблонным параметром, но невозможно специализировать отдельные функции-члены для шаблонов с несколькими шаблонными параметрами. Проиллюстрируем это утверждение примером.

```
template <class T> class Widget
{
   void Fun() { ... обобщенная реализация ... }
};
// OK: специализация функции-члена класса Widget
template <> Widget<char>::Fun()
{
   ... специализированная реализация ...
}
template <class T, class U> class Gadget
{
   void Fun() { ... обобщенная реализация ... }
```

```
};
// Ошибка! частичная специализация функции-члена
// класса Gadget невозможна!
template <class U> void Gadget<char, U>::Fun()
{
   { ... специализированная реализация ... }
};
```

3. *Автор библиотеки не может задать несколько значений по умолчанию.* В лучшем случае программист, реализующий шаблонный класс, может задать для каждой функции-члена одну реализацию по умолчанию. Шаблонная функция-член не может иметь *несколько* аргументов, заданных по умолчанию.

Сравним список недостатков множественного наследования со списком недостатков шаблонов. Интересно, что множественное наследование и шаблоны дополняют друг друга. Множественное наследование устроено довольно примитивно; шаблоны, наоборот, обладают богатыми возможностями специализации. Множественное наследование теряет информацию о типах; шаблоны ими изобилуют. Специализация шаблонов, в отличие от множественного наследования, не масштабируется. В шаблонах для функции-члена можно задать только один аргумент по умолчанию и создать неограниченное количество базовых классов.

Все это позволяет предположить, что с помощью комбинирования шаблонов и множественного наследования можно сконструировать очень гибкий механизм, пригодный для создания библиотек, содержащих готовые элементы программного обеспечения.

1.5. Стратегии и классы стратегий

Стратегии и классы стратегий позволяют реализовать безопасные, эффективные и легко настраиваемые элементы проектных решений. *Стратегия* (policy) определяет интерфейс обычного или шаблонного класса. Этот интерфейс состоит из определения внутренних типов, функций-членов и переменных-членов.

Понятие стратегии имеет много общего с *характеристиками* (traits) (Alexandrescu, 2000a), но отличается тем, что в них меньше внимания уделяется типам и больше — поведению. Кроме того, понятие стратегии, предложенное нами, напоминает *стратегии проектирования* (Gamma et al., 1995), но в отличие от них классы стратегий связываются на этапе компиляции.

Например, определим стратегию для создания объектов. Стратегия `Creator` описывает шаблонный класс типа `T`. Этот шаблонный класс должен предоставить функцию-член с именем `Create`, не имеющую аргументов и возвращающую указатель на объект класса `T`. Это означает, что каждый вызов функции `Create` должен возвращать указатель на новый объект класса `T`. Точный режим создания объекта определяется во время реализации стратегии.

Определим несколько классов стратегий, реализующих стратегию `Creator`. Для этого существует три способа. Во-первых, можно использовать оператор new. Во-вторых, вместо оператора new можно вызвать функцию `malloc` (Meyers, 1998b). В-третьих, новые объекты можно создавать, клонируя их прототипы. Рассмотрим эти методы.

```
template <class T>
struct OpNewCreator
{
   static T* Create()
   {
      return new T;
```

```
    }
};

template <class T>
struct MallocCreator
{
   static T* Create()
   {
      void* buf = std::malloc(sizeof(T));
      if (!buf) return 0;
      return new(buf) T;
   }
};

template <class T>
struct PrototypeCreator
{
   PrototypeCreator(T* pObj = 0)
      :pPrototype_(pObj)
   {}
   T* Create()
   {
      return pPrototype_ ? pPrototype_->Clone() : 0;
   }
   T* GetPrototype() { return pPrototype_; }
   void SetPrototype(T* pObj) {pPrototype_ = pObj;}
private:
   T* pPrototype_;
};
```

Данная стратегия может иметь сколько угодно реализаций. Реализации стратегий называются *классами стратегий*[2]. Классы стратегий по отдельности не используются. Они либо наследуются другими классами, либо содержатся в них.

Следует отметить, что, в отличие от классических интерфейсов (коллекций чисто виртуальных функций), интерфейсы стратегий не имеют четкого определения. Стратегии ориентируются на синтаксис, а не сигнатуру. Иными словами, стратегия Creator определяет, какая синтаксическая структура класса считается правильной, и не уточняет, какие именно функции класс должен реализовывать. Например, стратегия Creator не указывает, должна ли функция Create быть статической или виртуальной. Она лишь требует, чтобы шаблонный класс определял функцию-член Create. Кроме того, стратегия Creator выражает пожелание (но *не требует*), чтобы функция Create возвращала указатель на новый объект. Следовательно, вполне допускается, что в отдельных случаях функция-член Create может возвращать нуль или генерировать исключительную ситуацию.

Для одной стратегии можно реализовать несколько классов. Все они должны соответствовать интерфейсу, определенному стратегией. Затем, как мы вскоре убедимся, пользователь сам выберет, какой класс стратегии следует использовать в более крупных структурах.

Три класса стратегий, определенных выше, имеют разные реализации и даже слегка отличающиеся интерфейсы (например, класс PrototypeCreator имеет две дополнительные функции — GetPrototype и SetPrototype). Однако все они определяют

---

[2] Это название немного неточное, поскольку, как мы вскоре убедимся, реализациями стратегии могут быть и *шаблонные классы*.

функцию с именем Create, возвращающую значение требуемого типа, что соответствует требованиям стратегии Creator.

Посмотрим, как можно создать класс, применяющий стратегию Creator. Такой класс должен либо содержать три ранее определенных класса, либо наследовать их свойства.

```
// Код библиотеки
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{
   ...
};
```

Классы, использующие одну или несколько стратегий, называются *главными классами*[3] (host classes). В приведенном выше примере класс WidgetManager является главным классом, обладающим одной стратегией. Главные классы объединяют структуры и функциональные возможности своих стратегий в один составной модуль.

При конкретизации (instantiation) шаблона WidgetManager клиент передает ему искомую стратегию.

```
// Код приложения
typedef WidgetManager < OpNewCreator<Widget> > MyWidgetMgr;
```

Проанализируем возникший контекст. Если объекту класса MyWidgetMgr нужно создать объект класса Widget, он вызывает функцию Create() из подобъекта стратегии OpNewCreator<Widget>. Однако выбор стратегии создания объектов находится в компетенции пользователя класса WidgetManager. Этот класс по определению позволяет своим пользователям уточнять специфические аспекты своих функциональных возможностей.

Вот в чем заключается суть проектирования классов на основе стратегий.

1.5.1. Реализация классов стратегий с помощью шаблонных параметров

Как и в предыдущем примере, шаблонный аргумент стратегии зачастую излишен. Необходимость явно задавать шаблонный аргумент стратегии OpNewCreator создает неудобство. Обычно главный класс знает заранее или легко может установить шаблонный аргумент класса стратегии. В рассмотренном выше примере класс WidgetManager всегда управляет объектами, имеющими тип Widget, поэтому совершенно излишне и потенциально небезопасно требовать, чтобы пользователь снова указывал его при конкретизации стратегии OpNewCreator.

В этом случае код библиотеки может использовать *шаблонные шаблонные параметры* (template template parameters) следующим образом.

```
// Код библиотеки
template <template <class Created> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
};
```

Несмотря на внешний вид символ Created не относится к определению класса WidgetManager. Его нельзя использовать внутри класса WidgetManager — он пред-

[3] Несмотря на то что с технической точки зрения главные классы являются *шаблонными*, мы будем придерживаться данного выше определения, поскольку и главные классы, и шаблонные главные классы представляют собой одно и то же понятие.

ставляет собой формальный аргумент стратегии CreationPolicy (а не класса WidgetManager), поэтому его можно просто проигнорировать.

В коде приложения достаточно задать имя шаблона в конкретизации класса WidgetManager.

```
// Код приложения
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

Использование шаблонных шаблонных параметров вместе с классами стратегий — не просто вопрос удобства. Очень важно, что главный класс имеет доступ к шаблону, имея возможность конкретизировать его другим типом. Допустим, что объекту класса WidgetManager также нужно создать объект типа Garget, используя ту же стратегию создания объектов. Тогда соответствующий код может выглядеть следующим образом.

```
// Код библиотеки
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
   void DoSomething()
   {
      Gadget* pW = CreationPolicy<Gadget>().Create();
      ...
   }
};
```

Дает ли использование стратегий какие-либо преимущества? На первый взгляд немного. Все реализации стратегии Creator тривиальны. Автор класса WidgetManager мог просто написать код, предназначенный для создания объектов, в виде подставляемой функции, не прибегая к шаблону.

Однако использование стратегий придает классу WidgetManager необычайную гибкость. Во-первых, при конкретизации класса WidgetManager стратегии можно изменять *извне* так же легко, как и шаблонные аргументы. Во-вторых, программист может задавать свои собственные стратегии, характерные для конкретного приложения. Можно использовать оператор new, функцию malloc, прототипы или особые библиотеки управления памятью, присущие только данной операционной системе. *Ситуация выглядит так, будто класс* WidgetManager *представляет собой небольшое устройство для автоматической генерации кода, а пользователь задает способ, которым он генерирует код.*

Для того чтобы облегчить жизнь разработчикам прикладных программ, автор класса WidgetManager может определить набор часто применяемых стратегий и указать наиболее популярную стратегию в виде значения шаблонного аргумента, заданного по умолчанию.

```
template <template <class> class CreationPolicy = OpNewCreator>
class WidgetManager ...
```

Отметим, что стратегии значительно отличаются от виртуальных функций, которые обещают тот же эффект. Обычно программист, реализующий класс, определяет функции высокого уровня через элементарные виртуальные функции (primitive virtual function) и дает пользователю возможность замещать их поведение. Однако, как показано выше, стратегии основаны на более подробной информации о типах и статическом связывании. Эти два фактора представляют собой важные элементы проектирования, которое насыщено правилами, определяющими способ взаимодействия типов *до запуска программы на выполнение* и диктующими, что можно делать, а что — нет. Стратегии позволяют генерировать проекты, комбинируя варианты, не вникая в подробности их внутреннего устройства (in a typesafe manner). Кроме того, поскольку связывание главного класса с его стратегиями

осуществляется во время компиляции, компактность и эффективность полученного кода сравнима с его эквивалентами, разработанными вручную.

Разумеется, свойства стратегий делают их неудобными для динамического связывания и бинарных интерфейсов, поэтому, по существу, интерфейсы и стратегии дополняют друг друга, а не конкурируют.

1.5.2. Реализация классов стратегий с помощью шаблонных функций-членов

Вместо использования шаблонных шаблонных параметров можно применять шаблонные функции-члены и обычные классы. Это значит, что реализация стратегии представляет собой простой класс (в противоположность шаблонному классу), содержащий один или несколько шаблонных членов.

Например, можно переопределить стратегию `Creator` и задать обычный (нешаблонный) класс, содержащий шаблонную функцию `Create<T>`. Этот класс будет выглядеть следующим образом.

```
struct OpNewCreator
{
   template <class T>
   static T* Create()
   {
      return new T;
   }
};
```

Преимущество такого способа определения и реализации стратегии заключается в том, что он хорошо поддерживается старыми компиляторами. С другой стороны, стратегии, определенные таким образом, часто труднее объяснять, определять, реализовывать и применять.

1.6. Расширенные стратегии

Стратегия `Creator` описывает только одну функцию-член — `Create`. Однако в классе `PtototypeCreator` содержится на две функции больше: `GetPrototype` и `SetPrototype`. Проанализируем возникающий при этом контекст.

Поскольку класс `WidgetManager` наследует свой класс стратегии, а функции `GetPrototype` и `SetPrototype` являются открытыми членами класса `PrototypeCreator`, эти две функции передаются классу `WidgetManager` и непосредственно доступны клиентам.

Однако класс `WidgetManager` обращается только к функции-члену `Create`. Это все, что ему нужно для обеспечения своих функциональных возможностей. В то же время пользователям доступен более богатый интерфейс.

Пользователь, применяющий стратегию `Creator`, основанную на прототипах, может написать следующий код.

```
typedef WidgetManager<PrototypeCreator>
   MyWidgetManager;
...
Widget* pPrototype = ...;
MyWidgetManager mgr;
mgr.SetPrototype(pPrototype);
... используем объект mgr ...
```

Если впоследствии пользователь решит применить другую стратегию создания объектов, компилятор точно определит точки, в которых используется интерфейс,

ориентированный на применение прототипов. Именно это и требуется от хорошо продуманного проекта.

Возникающий контекст очень привлекателен. Клиенты, которым нужны расширенные стратегии, могут извлекать выгоды из более широких функциональных возможностей, не изменяя базовых функциональных свойств класса-владельца. Не забывайте, что *пользователи* (но не библиотека) определяют, какую стратегию применяет класс. В отличие от обычных множественных интерфейсов, стратегии дают пользователю возможность добавлять новые функциональные возможности главного класса, не вдаваясь в подробности его устройства.

1.7. Деструкторы классов стратегий

Разрабатывая классы стратегий, следует учитывать одну важную деталь. Чаще всего при создании классов, производных от своих стратегий, главный класс использует открытый интерфейс. По этой причине пользователь может автоматически конвертировать главный класс в стратегию, а затем удалить этот указатель с помощью оператора `delete`. Если в классе стратегии не определен виртуальный деструктор, применение оператора `delete` к указателю на объект этого класса приводит к непредсказуемым последствиям.[4]

```
typedef WidgetManager<PrototypeCreator>
   MyWidgetManager;
...
MyWidgetManager wm;
PrototypeCreator<Widget>* pCreator = &wm; // Сомнительно, но можно.
delete pCreator; // Прекрасно компилируется,
                 // но приводит к непредсказуемым последствиям.
```

Однако определение виртуального деструктора для стратегии противоречит ее статической природе и снижает производительность. Многие стратегии вообще не содержат данных-членов, определяя, по сути, лишь функциональные возможности. Любая виртуальная функция приводит к избыточному размеру объектов данного класса, поэтому применения виртуального конструктора следует избегать.

Решение этой проблемы заключается в следующем. При выводе главного класса из классов стратегий следует использовать защищенное или закрытое наследование. Однако это может ограничить применение расширенных стратегий (раздел 1.6).

Подобную задачу можно легко и эффективно решить, если потребовать, чтобы стратегии применяли невиртуальный защищенный деструктор.

```
template <class T>
struct OpNewCreator
{
   static T* Create()
   {
      return new T;
   }
protected:
   ~OpNewCreator() {}
};
```

Поскольку деструктор защищен, объекты стратегий могут уничтожаться только объектами производных классов, поэтому теперь внешние классы не могут применять опе-

[4] В главе 4 показано, почему это происходит.

ратор delete к указателям на объекты стратегий. Деструктор не виртуален, поэтому дополнительных затрат памяти или замедления работы программы не происходит.

1.8. Факультативные возможности, предоставляемые неполной конкретизацией

Язык C++ придает стратегиям особую мощь, снабжая их интересным свойством. Если функция-член шаблонного класса никогда не используется, она не конкретизируется — компилятор вообще игнорирует ее, проверяя лишь синтаксические ошибки.[5]

Это дает главным классам возможность задавать и применять факультативные свойства классов стратегий. Например, определим функцию-член SwitchPrototype в классе WidgetManager.

```
// Код библиотеки
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
   void SwitchPrototype(Widget* pNewPrototype)
   {
      CreationPolicy<Widget>& myPolicy = *this;
      delete myPolicy.GetPrototype();
      myPolicy.SetPrototype(pNewPrototype);
   }
};
```

Возникающий контекст очень интересен.

- Если пользователь конкретизирует класс WidgetManager с помощью класса стратегии Creator, поддерживающей работу с прототипами, можно использовать функцию SwitchPrototype.
- Если пользователь конкретизирует класс WidgetManager с помощью класса стратегии Creator, не поддерживающей работу с прототипами, и попытается использовать функцию SwitchPrototype, во время компиляции возникнет ошибка.
- Если пользователь конкретизирует класс WidgetManager с помощью класса стратегии Creator, не поддерживающей работу с прототипами, и не пытается использовать функцию SwitchPrototype, программа считается правильной.

Все сказанное выше означает, что класс WidgetManager может использовать факультативные расширенные интерфейсы, но при этом продолжает правильно работать и с интерфейсами, обладающими более бедными возможностями, пока пользователь не попытается использовать определенные функции класса WidgetManager.

Автор класса WidgetManager может определить стратегию Creator следующим образом.

> Стратегия Creator определяет шаблонный класс типа T, содержащий функцию-член Create. Эта функция должна возвращать указатель на новый объект типа T. В реализации можно определить две дополнительные функции-члены:T* GetPrototype() и SetPrototype(T*), характеризующие семантику

[5] В соответствии со стандартом языка C++ строгость синтаксической проверки неиспользуемых шаблонных функций зависит от конкретной реализации. Компилятор не выполняет семантическую проверку — например, поиск символов не проводится.

получения и установки прототипов, используемых для создания объектов. В этом случае класс WidgetManager содержит функцию-член SwitchPrototype(T* pNewPrototype), удаляющую текущий прототип и задающую его для входного аргумента.

Вместе с классами стратегий неполная конкретизация обеспечивает замечательную свободу при разработке библиотек. Можно реализовать тощие (lean) главные классы, способные использовать дополнительные свойства и состоящие из минимальных стратегий.

1.9. Комбинирование классов стратегий

Наибольщую пользу приносят комбинации стратегий. Обычно хорошо разработанный класс использует несколько стратегий для реализации разных аспектов своего функционирования. Затем пользователь библиотеки выбирает нужный режим работы, комбинируя несколько классов стратегий.

Например, рассмотрим процесс разработки класса для обобщенного интеллектуального указателя. (Полная реализация этого класса осуществлена в главе 7.) Допустим, что нам нужно указать два варианта проектного решения, используя стратегии: потоковую модель и проверку перед разыменованием. Затем мы реализуем шаблонный класс SmartPtr, использующий две стратегии.

```
template
<
   class T,
   template <class> class CheckingPolicy,
   template <class> class TheadingModel
>
class SmartPtr;
```

Класс SmartPtr имеет три шаблонных параметра — тип указателя и две стратегии. Внутри этого класса эти две стратегии детально описаны, так что класс SmartPtr становится хорошо продуманной оболочкой, интегрирующей две разные стратегии, а не жесткой, законсервированной реализацией. Разрабатывая класс SmartPtr таким образом, можно предоставить пользователю возможность конфигурировать его с помощью оператора typedef.

```
typedef SmartPtr<Widget, NoChecking, SingleThreaded>
   WidgetPtr;
```

В рамках некоторых приложений можно определять и применять несколько классов интеллектуальных указателей.

```
typedef SmatPtr<Widget, EnforceNotNull, SingleThreaded>
   SafeWidgetPtr;
```

Две стратегии можно сформулировать следующим образом.

Стратегия Checking. Шаблонный класс CheckingPolicy<T> должен содержать функцию-член Check, которую можно вызывать с параметром типа T*. Перед разыменованием указателя класс SmartPtr вызывает функцию Check, передавая ей объект, на который ссылается данный указатель.

Стратегия ThreadingModel. Шаблонный класс ThreadingModel<T> должен содержать внутренний тип, называемый Lock, конструктор которого получает ссылку T&. На протяжении всего существования объекта Lock все операции над объектами типа T регистрируются.

Например, ниже приведена реализация классов стратегий NoChecking и EnforceNotNull.

```
template <class T> struct NoChecking
{
   static void Check(T*) {}
};
template <class T> struct EnforceNotNull
{
   class NullPointerException : public std::exception { ... }
   static void Check(T* ptr)
   {
      if (!ptr) throw NullPointerException();
   }
};
```

Подключая разные классы стратегии проверки, можно реализовать различные режимы работы объектов. Ниже показано, что можно даже инициализировать объект, на который ссылается указатель, значением, заданным по умолчанию, получив ссылку на указатель.

```
template <class T> struct EnsureNotNull
{
   static void Check(T*& ptr)
   {
      if (!ptr) ptr = GetDefaultValue();
   }
};
```

Класс SmartPtr использует стратегию Checking следующим образом.

```
template
<
   class T,
   template <class> class CheckingPolicy,
   template <class> class ThreadingModel
>
class SmartPtr
   : public CheckingPolicy<T>,
     public ThreadingModel<SmartPtr>
{
   ...
   T* operator->()
   {
      typename ThreadingModel<SmartPtr>::Lock guard(*this);
      CheckingPolicy<T>::Check(pointee_)
      return pointee_;
   }
private:
   T* pointee_;
};
```

Отметим, что оба класса стратегий CheckingPolicy и ThreadingModel используют одни и те же функции. Функционирование оператора SmartPtr::operator-> зависит от двух шаблонных аргументов. В этом проявляется сила, присущая комбинированию стратегий.

Разложив класс на ортогональные стратегии, можно обеспечить широкий спектр режимов работы с помощью небольшого по объему кода.

1.10. Настройка структур с помощью классов стратегий

Одно из ограничений, присущих шаблонам, как было указано в разделе 1.4, состоит в том, что с помощью шаблонов невозможно уточнить структуру класса — только его режим работы. В то же время проектирование, основанное на использовании стратегий, позволяет уточнить структуру вполне естественным образом.

Допустим, что нам нужно создать класс SmartPtr без применения указателей. Например, на отдельных платформах некоторые указатели могут быть представлены особым образом — в виде целого числа, которое передается системной функции, возвращающей настоящий указатель. Для того чтобы решить эту задачу, можно определить доступ к указателю, например, с помощью стратегии Structure. Эта стратегия абстрагирует способ хранения указателя. Следовательно, в этой стратегии нужно указать типы с именем PointerType (тип указателя) и ReferenceType (тип ссылки).

То, что тип указателя не задан жестко как T*, представляет собой большое преимущество. Например, можно использовать класс SmartPtr с нестандартными типами указателей (такими, как указатели near и far, применяющиеся на компьютерах, имеющих сегментированную архитектуру) или легко реализовать такие остроумные решения, как функции before и after (Stroustrup, 2000a). Эти возможности чрезвычайно интересны.

По умолчанию интеллектуальный указатель хранится как простой указатель, снабженный интерфейсом стратегии Structure.

```
template <class T>
class DefaultSmartPtrStorage
{
public:
   typedef T* PointerType;
   typedef T& ReferenceType;
protected:
   PointerType GetPointer() { return ptr_; }
   void SetPointer(PointerType ptr) { pointee_ = ptr; }
private:
   PointerType ptr_;
};
```

Фактический способ хранения полностью скрыт за фасадом интерфейса стратегии Structure. Теперь класс Smartptr может использовать стратегию хранения указателя Storage, а не группироваться с типом T*.

```
template
<
   class T,
   template <class> class CheckPolicy,
   template <class> class ThreadingModel,
   template <class> class Storage = DefaultSmartPtrStorage
>
class SmartPtr;
```

Разумеется, для того, чтобы внедриться в нужную структуру, класс SmartPtr должен либо быть производным от класса Storage<T>, либо группироваться с объектом типа Storage<T>.

1.11. Совместимые и несовместимые стратегии

Допустим, нам нужны две конкретизации класса SmartPtr: FastWidgetPtr, указатель без проверки, и SafeWidgetPtr, указатель с проверкой перед разыменованием.

Возникает интересный вопрос: нужно ли присваивать объекты класса FastWidgetPtr объектам класса SafeWidgetPtr? Следует ли присваивать объекты этих классов другим переменным? Если такие преобразования необходимы, как их реализовать?

Отталкиваясь от той точки зрения, что указатели типа SafeWidgetPtr накладывают больше ограничений, чем указатели типа FastWidgetPtr, естественно допустить преобразование типа FastWidgetPtr в тип SafeWidgetPtr. Причина этого заключается в том, что язык С++ уже поддерживает неявные преобразования, увеличивающие ограничения, например, переменных неконстантных типов — в константные.

С другой стороны, свободное преобразование объектов класса SafeWidgetPtr в объекты класса FastWidgetPtr опасно, поскольку в приложениях основная масса кода может использовать указатели типа SafeWidgetPtr, и только небольшое ядро, для которого скорость выполнения является крайне важной, может работать с указателями типа FastWidgetPtr. Разрешая только явные, контролируемые преобразования в тип FastWidgetPtr, можно свести к минимуму использование указателей этого типа.

Наилучшим и наиболее масштабируемым способом взаимного преобразования стратегий является инициализация и копирование объектов класса SmartPtr *стратегия за стратегией* (policy by policy), как показано ниже. (Для простоты рассмотрена только одна стратегия — Checking.)

```
template
<
   class T,
   template <class> class CheckingPolicy
>
class SmartPtr : public CheckingPolicy<T>
{
   ...
   template
   <
      class T1,
      template <class> class CP1
   >
   SmartPtr(const SmartPtr<T1, CP1>& other)
      : pointee_(other.pointee_), CheckingPolicy<T>(other)
   { ... }
};
```

Класс SmartPtr реализует шаблонную копию конструктора, получающую другую конкретизацию класса SmartPtr. Код, выделенный полужирным шрифтом, инициализирует компоненты класса SmartPtr компонентами другого класса SmartPtr<T1, CP1>, полученными в виде аргументов.

Вот как работает этот способ. (Проследим за выполнением кода конструктора.) Допустим, у нас есть класс ExtendedWidget, производный от класса Widget. Если объект класса SmartPtr<Widget, NoChecking> проинициализировать объектом класса SmartPtr<ExtendedWidget, NoChecking>, компилятор попытается проинициализировать указатель Widget* указателем ExtendedWidget* (что вполне допустимо), а объект класса NoChecking — объектом класса SmartPtrExtended<ExtendedWidget, NoChecking>. Возможно, это выглядит подозрительно, но не забывайте, что класс SmartPtr является производным от своих стратегий, так что, по существу, компилятор может легко распознать, что мы инициализируем объект класса NoChecking другим объектом того же класса. Таким образом, процесс инициализации выполняется без проблем.

Перейдем теперь к более интересной части. Допустим, нам нужно инициализировать объект класса SmartPtr<Widget, EnforceNotNull> объектом класса SmartPtr<ExtendedWidget, NoChecking>. Как и раньше, указатель типа ExtendedWidget* без проблем преобразовывается в указатель типа Widget*. Затем компилятор попытается сравнить конструкторы классов SmartPtr<ExtendedWidget, NoChecking> и EnforceNorNull.

Если класс EnforceNorNull реализует конструктор, получающий объект NoChecking, то компилятор проверяет этот конструктор. Если класс NoChecking реализует оператор преобразования в тип EnforceNull, вызывается функция, реализующая это преобразование. В остальных случаях возникает ошибка компиляции.

Очевидно, что при реализации преобразований стратегий друг в друга возникает удвоенная гибкость: преобразование можно реализовать как с помощью конструктора, так и с помощью оператора.

По сложности это напоминает оператор присваивания, однако, к счастью, Sutter (2000) описал очень остроумный способ, позволяющий реализовать оператор присваивания с помощью конструктора копирования. (Он настолько остроумен, что о нем *обязательно* следует прочитать. Этот способ был применен при реализации класса SmartPtr в библиотеке Loki.)

Несмотря на то что преобразования объектов классов NoChecking в объекты класса EnforceNotNull и даже наоборот имеют довольно ясный смысл, некоторые преобразования совершенно бессмысленны. Представьте себе преобразование указателя, подсчитывающего ссылки, в указатель, поддерживающий другую стратегию владения (ownership policy), например, уничтожение после копирования (destructive copy) наподобие std::auto_ptr. Такое преобразование семантически неверно. Определение указателя, подсчитывающего ссылки, состоит в следующем: все указатели на один и тот же объект считаются известными и отслеживаются с помощью единственного счетчика. При попытке применить другую стратегию происходит нарушение инварианта, гарантирующего правильную работу указателя, подсчитывающего ссылки.

В заключение заметим, что неявные преобразования, изменяющие стратегию владения, применять не следует и обращаться с ними нужно с максимальной осторожностью. Лучше всего изменять стратегию владения указателем, подсчитывающим ссылки, явно вызывая соответствующую функцию. Эта функция будет успешно работать, только если счетчик ссылок исходного указателя равен 1.

1.12. Разложение классов на стратегии

При проектировании, основанном на применении стратегий, самое трудное — правильно разложить функциональные возможности класса на стратегии. Основное правило заключается в следующем: нужно идентифицировать и назвать проектные решения, определяющие режим работы класса. Все, что можно сделать несколькими способами, должно идентифицироваться и передаваться от класса к стратегии. Не забудьте: проектные решения, замурованные в класс, подобны константам, "зашитым" в код.

Рассмотрим, например, класс WidgetManager. Если внутри класса WidgetManager создается новый объект класса Widget, процесс создания объекта нужно передать стратегиям. Если класс WidgetManager хранит коллекцию объектов класса Widget, имеет смысл выразить этот способ хранения в виде стратегии, если при этом какой-то специфический механизм хранения не имеет особого приоритета.

В идеале главный класс полностью описывается с помощью отдельных внутренних стратегий. Он делегирует все проектные решения и ограничения стратегиям. Такой класс представляет собой оболочку коллекции стратегий и только дирижирует стратегиями, добиваясь нужного режима работы.

Недостаток обобщенного главного класса, чрезмерно перегруженного стратегиями, заключается в слишком большом количестве шаблонных параметров. На практике, если количество параметров больше четырех, с классом становится трудно работать. Тем не менее они оправдывают свое существование, если главный класс обеспечивает сложные и полезные функциональные возможности.

Определение класса с помощью оператора typedef представляет собой важный инструмент, позволяющий использовать классы, основанные на стратегиях. Применение этого оператора — не просто вопрос удобства. Он гарантирует упорядоченное использование класса и его легкое сопровождение. Например, рассмотрим следующее определение типа.

```
typedef SmartPtr
<
   Widget,
   RefCounted,
   NoChecked
>
WidgetPtr;
```

В программе было бы довольно затруднительно использовать длинную специализацию класса SmartPtr вместо класса WidgetPtr. Однако утомительность программирования ничего не значит по сравнению с главными проблемами, заключающимися в *понимании* и *сопровождении* программы. В процессе проектирования определение класса Widget может измениться — например, может применяться стратегия проверки, отличающаяся от стратегии NoChecked. Очень важно, что класс WidgetPtr использует вся программа, а не только жестко закодированная конкретизация класса SmartPtr. Это напоминает различие между вызываемыми и подставляемыми (inline) функциями. С технической точки зрения подставляемая функция решает ту же задачу, однако создать абстракцию на ее основе невозможно.

Раскладывая класс на стратегии, важно найти *ортогональное* разложение (orthogonal decomposition). Возникающие при этом стратегии совершенно не зависят друг от друга. Распознать неортогональное разложение очень легко — в нем разные стратегии нуждаются в информации друг о друге.

Например, представим себе стратегию Array в интеллектуальном указателе. Эта стратегия очень проста — она диктует, ссылается интеллектуальный указатель на массив или нет. Можно определить стратегию Array так, чтобы она содержала функцию-член T& ElementAt(T* ptr, unsigned int index), а также аналогичную версию для типа const T. В стратегиях, не работающих с массивами, функция ElementAt просто не определяется, поэтому попытка ее использовать приведет к ошибке во время компиляции. Функция ElementAt, в соответствии с определением, данным в разделе 1.6, представляет собой факультативную расширенную функциональную возможность.

Реализации двух классов, основанных на стратегии Array, приведены ниже.

```
template <class T>
struct IsArray
{
   T& ElementAt(T* ptr, unsigned int index)
```

```
    {
        return ptr[index];
    }
    const T& ElementAt(T* ptr, unsigned int index) const
    {
        return ptr[index];
    }
};
template <class T> struct IsNotArray {};
```

Проблема заключается в том, что предназначение стратегии Array (указать, ссылается интеллектуальный указатель на массив или нет) плохо согласованно с другой стратегией — разрушением. Уничтожать указатели на объекты можно с помощью оператора delete, а указатели на массивы, состоящие из объектов, — оператором delete[].

Две независимые друг от друга стратегии являются ортогональными. На основании данного определения можно утверждать, что стратегии Array и Destroy не являются ортогональными.

Для того чтобы описать способ работы с величинами, хранящимися в массиве, и способ уничтожения объектов в виде отдельных стратегий, необходимо описать их взаимодействие. Например, в стратегии Array, кроме функций, можно предусмотреть булевскую константу и передать ее в стратегию Destroy. Это осложняет и несколько ограничивает процесс проектирования обеих стратегий.

Неортогональные стратегии несовершенны, поэтому нужно стараться их избегать. Они снижают уровень типовой безопасности во время компиляции и усложняют процесс разработки как главного класса, так и классов стратегий.

Если приходится прибегать к неортогональным стратегиям, нужно хотя бы минимизировать их взаимную зависимость, передавая класс стратегии в качестве аргумента шаблонной функции другого класса стратегии. Этот способ компенсирует недостатки, связанные с неортогональностью, за счет гибкости, присущей интерфейсам, основанным на шаблонах. Теневой стороной этого метода остается тот факт, что одна стратегия должна явно задавать некоторые детали реализации других стратегий. Это снижает степень инкапсуляции.

1.13. Резюме

Проектирование — это выбор. Чаще всего проблема заключается не в том, что задачу в принципе невозможно решить, а в том, что у нее существует множество способов решения. Необходимо знать, какие из возможных решений удовлетворительно решают поставленную задачу. Это вынуждает нас переходить от высших архитектурных уровней к низшим. Более того, выбранные варианты можно комбинировать, что приводит к появлению “проклятия выбора” из большого количества вариантов.

Для того чтобы преодолеть “проклятие выбора” с помощью кода, имеющего разумные размеры, разработчики библиотек, предназначенных для проектирования программного обеспечения, должны изобретать и применять специальные способы. Изначально эти методы были предназначены для обеспечения гибкости при автоматической генерации кода в сочетании с небольшим количеством элементарных механизмов (primitive devices). Сами по себе библиотеки предоставляют огромное количество таких механизмов. Более того, они содержат *спецификации* (specifications), на основе которых создаются новые механизмы, так что пользователь может создавать их самостоятельно. Это существенно влияет на открытость (open-ended) проектирования,

основанного на стратегиях. Эти механизмы называются *стратегиями*, а их реализации, соответственно, называются *классами стратегий*.

Механизм стратегии основан на комбинации шаблонов и множественного наследования. Класс, использующий стратегии (главный класс), представляет собой шаблон со многими шаблонными параметрами (часто называемыми *шаблонными шаблонными параметрами*), каждый из которых является стратегией. Главный класс "переадресовывает" части функциональных возможностей стратегиям и действует в качестве хранилища нескольких согласованных между собой стратегий.

Классы, построенные на основе стратегий, обладают богатыми возможностями и элегантно раскладываются на функциональные свойства. Стратегия может обеспечивать функциональное свойство, передаваемое главному классу с помощью открытого наследования. Более того, главный класс может реализовывать расширенные функциональные возможности, которые используют факультативные свойства стратегий. Если факультативные функциональные возможности не предусмотрены, главный класс компилируется успешно при условии, что расширенные функциональные возможности не используются.

Мощь стратегий проявляется через их способность смешиваться и настраиваться. Класс, основанный на применении стратегий, может реализовывать различные режимы работы, комбинируя более простые виды поведения, предусмотренные стратегиями. Это превращает стратегии в эффективное средство против "проклятия выбора".

Используя классы стратегий, можно настраивать не только режим работы, но и структуру. Это важное свойство выводит проектирование, основанное на стратегиях, за рамки простого генерирования типов, характерного для контейнерных классов.

Классы, основанные на стратегиях, обеспечивают гибкость преобразований. При использовании копирования "стратегия за стратегией" каждая стратегия может распознавать, к каким еще стратегиям она имеет доступ, или преобразовываться в них с помощью соответствующего конструктора преобразования, оператора преобразования или обоих одновременно.

Разбивая классы на стратегии, нужно следовать двум важным принципам. Первый из них — следует локализовать, назвать и изолировать проектные решения в классе. Эта цель является предметом компромисса и может достигаться различными путями. Второй принцип — нужно искать ортогональные стратегии, т.е. стратегии, не нуждающиеся во взаимодействии друг с другом и способные изменяться независимо друг от друга.

2

Приемы программирования

В этой главе описывается множество приемов программирования на языке C++, используемых на протяжении всей книги. Поскольку эти приемы оказываются полезными в разных ситуациях, они описываются в максимально общем виде, чтобы их легко было применить повторно. Таким образом, один и тот же прием может использоваться в разных ситуациях. Некоторые из этих приемов, например, частичная специализация шаблонов, представляют собой особенности языка программирования. Другие, например, диагностические утверждения на этапе компиляции (compile-time assertions), программируются отдельно.

В этой главе рассматриваются следующие приемы и инструменты программирования.

- Статическая проверка диагностических утверждений.
- Частичная специализация шаблонов.
- Локальные классы.
- Отображения между типами и значениями (шаблонные классы `Int2Type` и `Type2Type`).
- Шаблонный класс `Select`, средство для выбора типа на этапе компиляции на основе проверки логического выражения.
- Распознавание конвертируемости и наследования на этапе компиляции.
- Класс `TypeInfo`, удобная оболочка для класса `std::type_info`.
- Класс `Traits`, коллекция характеристик (traits), применимых к любому типу в языке C++.

Взятые по отдельности, каждый прием и соответствующий ему код могут выглядеть тривиально. Обычно их размер не превышает 5–10 вполне понятных строк. Однако эти приемы программирования обладают важной особенностью: они "нетерминальны", т.е. их можно комбинировать друг с другом, генерируя идиомы высокого уровня. В совокупности они образуют солидную основу для служебной библиотеки, позволяющей создавать мощные архитектурные конструкции.

Приемы иллюстрируются примерами, а не сухим описанием. Читая другие главы книги, вы можете возвращаться к этой главе за справками.

2.1. Статическая проверка диагностических утверждений

С появлением обобщенного программирования на языке С++ возникла необходимость в более совершенных средствах статической проверки (и более конкретных сообщениях об ошибках).

Допустим, что вы разрабатываете функцию для безопасного приведения типов (safe casting). Вам нужно привести один тип к другому, не потеряв при этом информацию, причем типы большего размера не должны преобразовываться в типы меньшего размера.

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
   assert(sizeof(From) <= sizeof(To));
   return reinterpret_cast<To>(from);
}
```

Эта функция вызывается с помощью точно такой же синтаксической конструкции, как и встроенное приведение типов в языке С++.

```
int i = ...;
char* p = safe_reinterpret_cast<char*>(i);
```

Шаблонный аргумент `To` указывается явно, а компилятор самостоятельно выводит шаблонный аргумент `From`, анализируя тип переменной `i`. Сравнивая размеры, можно убедиться, что тип, к которому выполняется приведение, способен хранить все биты величины приводимого типа. Таким образом, указанный выше код либо выполняет вроде бы правильное приведение типов[1], либо проверяет диагностическое утверждение в ходе выполнения программы.

Очевидно, было бы гораздо удобнее обнаруживать такие ошибки на этапе компиляции. Прежде всего, приведение типов может находиться в очень редко используемой ветви программы. При переносе приложения на новый компилятор или платформу невозможно запомнить все машинно-зависимые части программы, поэтому в приложении могут остаться скрытые ошибки, которые приведут к ее краху прямо на глазах у вашего заказчика.

Однако все не так безнадежно. Выражения, подлежащие вычислению, являются статическими константами (compile-time constant). Это означает, что их проверку можно осуществлять на этапе компиляции, а не в ходе выполнения программы. Идея этого подхода заключается в следующем. Компилятору передается языковая конструкция, являющаяся допустимой, если значение выражения не равно нулю, и недопустимой в противном случае. Таким образом, получив выражение, значение которого равно нулю, компилятор выдаст сообщение об ошибке.

Простейшее решение задачи статической проверки диагностических утверждений (Van Horn, 1997) одинаково хорошо работает как в языке С, так и в языке С++, и основано на том, что массивы нулевой длины запрещены.

```
#define STATIC_CHECK(expr) { char unnamed[(expr) ? 1 : 0]; }
```

Теперь напишем следующий фрагмент.

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
```

[1] В большинстве компьютеров, т.е., используя функцию `reinterpret_cast`, никогда нельзя быть уверенным в успехе.

```
    STATIC_CHECK(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

Если размер системного указателя превышает размер символа, компилятор выдаст сообщение, что программа пытается создать массив нулевой длины.

С этим подходом связана одна проблема — сообщение об ошибке, выдаваемое компилятором, малоинформативно. "Невозможно создать массив нулевой длины" совсем не означает, что "Тип char слишком узок для указателя". Сделать сообщения об ошибках настраиваемыми и платформно-независимыми очень трудно. Для сообщений об ошибках не существует правил. Все они характерны лишь для данного компилятора. Например, если ошибка относится к неопределенной переменной, имя этой переменной в сообщении может не указываться.

Гораздо лучше положиться на шаблоны, имеющие информативные имена. К счастью, компилятор обязан указать имя такого шаблона в сообщении об ошибке.

```
template<bool> struct CompileTimeError;
template<> struct CompileTimeError<true> {};

#define STATIC_CHECK(expr) \
    (CompileTimeError<(expr) != 0>())
```

Структура CompileTimeError является шаблоном, получающим параметр, не являющийся типом (булевскую константу). Она определена только для значения true этой булевской константы. Если попытаться конкретизировать шаблон с помощью выражения CompileTimeError<false>, компилятор выдаст примерно такое сообщение: "Неопределенная специализация CompileTimeError<false>". Это сообщение немного содержательнее предыдущего и говорит о том, что ошибка сделана преднамеренно.

Разумеется, здесь есть простор для совершенствования. Можно ли настраивать сообщения об ошибках? Идея состоит в следующем. Нужно передать дополнительный параметр в шаблон STATIC_CHECK и каким-то образом сделать так, чтобы он появился в сообщении об ошибке. К недостаткам этого приема следует отнести тот факт, что настраиваемое сообщение об ошибке должно подчиняться правилам образования идентификаторов в языке C++ (не содержать пробелов, не начинаться с цифры и т.д.). Это приводит к улучшенной версии шаблона CompileTimeError, показанной ниже. Фактически шаблон CompileTimeError в новом контексте становится малопонятным. Более осмысленным является шаблон CompileTimeChecker.

```
template<bool> struct CompileTimeChecker;
{
    CompileTimeChecker(...);
};
template<> struct CompileTimeChecker<false> {};
#define STATIC_CHECK(expr, msg) \
   {\
    class ERROR_##msg {}; \
    (void)sizeof(CompileTimeChecker<(expr) != 0> ((ERROR_##msg())));\
   }
```

Предположим, что sizeof(char) < sizeof(void*). (Стандарт языка не гарантирует, что это выражение обязательно является истинным.) Посмотрим, что произойдет, если написать следующий фрагмент программы.

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
   STATIC_CHECK(sizeof(From) <= sizeof(To),
      Destination_Type_Too_Narrow);
   return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

После обработки макроса препроцессором код функции safe_reinterpret_cast примет следующий вид.

```
template <class To, class from>
To safe_reinterpret_cast(From from)
{
   {
     class ERROR_Destination_Type_Too_Narrow {};
     (void)sizeof(
       CompileTimeChecker<(sizeof(From)) <= sizeof(To))>(
          ERROR_Destination_Type_Too_Narrow ()));
   }
   return reinterpret_cast<To>(from);
}
```

Этот код определяет *локальный класс* (local class) ERROR_Destination_Type_Too_Narrow, имеющий пустое тело. Затем он создает временное значение типа CompileTimeChecker<sizeof(From) <= sizeof(To))>, инициализированное временным значением типа ERROR_Destination_Type_Too_Narrow. В заключение функция sizeof вычисляет размер возникшей временной переменной.

А теперь сделаем фокус! Специализация CompileTimeChecker<true> содержит конструктор, не имеющий аргументов. Это значит, что если выражение, проверяемое на этапе компиляции, имеет значение true, то полученная в результате программа считается правильной, в противном случае возникает ошибка компиляции. Компилятор не может выполнить преобразование из типа ERROR_Destination_Type_Too_Narrow в тип CompileTimeChecker<false>. Приятнее всего, что теперь компилятор выводит приблизительно такое сообщение об ошибке: "Ошибка: невозможно преобразовать тип ERROR_Destination_Type_Too_Narrow в тип CompileTimeChecker<false>".

Есть!

2.2. Частичная специализация шаблонов

Частичная специализация шаблонов позволяет специализировать шаблонный класс с помощью подмножества его возможных параметров конкретизации.

Рассмотрим шаблонный класс Widget.

```
template < class Window, class Controller>
class Widget
{
   .. обобщенная реализация ...
};
```

Применим полную явную специализацию шаблона.

```
template <>
class Widget <ModalDialog, MyController>
{
```

```
   .. специализированная реализация ...
};
```

Классы ModalDialog и MyController определяются в прикладной программе. Обнаружив определение специализации класса Widget, при определении объектов типа Widget<ModalDialog, MyController> компилятор будет применять специализированную, а в остальных случаях — обобщенную реализацию.

Однако иногда возникает необходимость специализировать класс Widget для *любых* конкретизаций класса Window и отдельной конкретизации класса MyController. Вот как выглядит частичная специализация шаблона в этом случае.

```
// Частичная специализация шаблонного класса Widget
template <class Window>
{
   ... частичная специализированная реализация ...
};
```

Обычно в частичной специализации шаблонного класса указываются лишь некоторые шаблонные аргументы, а остальные остаются обобщенными (generic). Конкретизируя шаблонный класс в программе, компилятор пытается найти оптимальное соответствие. Сложный и точный алгоритм поиска соответствий позволяет программисту каждый раз по-новому осуществлять частичную специализацию. Допустим, что шаблонный класс Button получает один шаблонный параметр. Затем, даже если класс Widget специализируется для любых конкретизаций класса Window и отдельной конкретизации класса MyController, класс Widget можно и дальше частично специализировать для *всех* конкретизаций класса Button и конкретизации MyController.

```
template <class ButtonArg>
class Widget<Button<ButtonArg>, MyController>
{
   ...
};
```

Как видим, возможности частичной шаблонной специализации поразительны. Конкретизируя шаблон, компилятор выполняет *сопоставление с эталоном* (pattern matching) существующих частичных и полных специализаций, подыскивая наилучший вариант. Это обеспечивает программисту невероятную свободу действий.

К сожалению, частичную шаблонную специализацию невозможно применить к функциям, независимо от того, являются они членами класса или нет. Это несколько снижает уровень гибкости и степень детализации программы.

- Несмотря на то что в шаблонном классе можно выполнять *полную шаблонную специализацию* функций-членов, к ним нельзя применять частичную шаблонную специализацию.
- Шаблонные функции из пространства имен (не являющиеся членами какого-либо класса) невозможно частично специализировать. Больше всего на частичную специализацию шаблонных функций из пространства имен (namespace-level template functions) похож процесс их замещения. С практической точки зрения это означает, что возможность детальной специализации существует лишь для параметров функций, но не для возвращаемых ими значений или используемых внутри нее типов.

```
template <class T, class U> T Fun(U obj); // Исходный шаблон
template <class U> void Fun<void, U>(U obj); // Неправильная
                                             // частичная специализация
template <class T> T Fun (Window obj); // Правильно (перегрузка)
```

Недостаточная детализация частичной специализации существенно облегчает жизнь разработчикам компиляторов, но затрудняет работу программистов. Некоторые из инструментов, описанных ниже (например, классы Int2Type и Type2Type), специально предназначены для преодоления этих ограничений частичной специализации.

Частичная специализация шаблонов в книге используется очень широко. Фактически все средства для работы со списками типов (глава 3) созданы с ее помощью.

2.3. Локальные классы

Локальные классы — это интересное и мало изученное свойство языка C++. Эти классы можно определять прямо внутри функций.

```
void Fun()
{
   class Local
   {
      ... функции-члены
      ... определения функций-членов ...
   };
   ... код, использующий локальный класс ...
}
```

Существует несколько ограничений — в локальных классах нельзя определять статические переменные-члены и обращаться к нестатическим локальным переменным. Особенно интересно, что локальные классы можно применять внутри шаблонных функций. Локальный класс, определенный внутри шаблонной функции, может использовать ее шаблонные параметры.

Шаблонная функция MakeAdapter в приведенном ниже коде адаптирует один интерфейс к другому. Функция MakeAdapter реализует интерфейс на лету с помощью локального класса. Локальный класс хранит члены обобщенного типа.

```
class Interface
{
   public:
      virtual void Fun() = 0;
      ...
};

template <class T, class P>
Interface* MakeAdapter(const T& obj, const P& arg)
{
   class Local : punlic Interface
   {
   public:
      Local(const T& obj, const P& arg)
         : obj_(obj), arg_(arg) {}
      virtual void Fun()
      {
         obj_.Call(arg_);
      }
   private:
      T obj_;
      P arg_;
   };
   return new Local(obj, arg);
}
```

Легко доказать, что любую идиому, использующую локальный класс, можно реализовать с помощью шаблонного класса, определенного вне функции. Иными словами, локальные классы для создания идиом непригодны. С другой стороны, локальные классы могут упростить реализации и улучшить локальность символов.

Однако локальные классы обладают уникальным свойством: они являются *терминальными* (final). Внешние пользователи не могут создать производные от них классы, скрытые внутри функции. Без локальных классов было бы невозможно добавить неименованное пространство имен в отдельный транслируемый модуль.

В главе 11 локальные классы используются для создания трамплинных функций.

2.4. Отображение целочисленных констант в типы

Простой шаблон, впервые описанный в работе (Alexandrescu, 2000b), может оказаться очень полезным для создания многих идиом обобщенного программирования.

```
template <int v>
struct Int2Type
{
   enum { value = v };
};
```

Класс `Int2Type` генерирует различные типы для разных значений передаваемой ему целочисленной константы. Это происходит потому, что у разных конкретизаций шаблона разные типы. Таким образом, тип конкретизации `Int2Type<0>` отличается от типа конкретизации `Int2Type<1>` и т.д. Кроме того, значение, генерирующее тип, "запоминается" членом `value` перечисления `enum`.

Класс `Int2Type` применяется, если нужно быстро "типизировать" целочисленную константу. С его помощью можно выбирать разные функции в зависимости от результата вычислений, выполняемых на этапе компиляции. Фактически этот класс обеспечивает статическую диспетчеризацию (compile-time dispatching) константных значений целочисленного типа.

Обычно класс `Int2Type` используется, когда одновременно выполняются два условия.

- Нужно вызвать одну из нескольких функций в зависимости от значения статической константы.
- Эту диспетчеризацию нужно выполнить на этапе компиляции.

Для осуществления диспетчеризации во время выполнения программы (run-time dispatching) можно применять условный оператор `if-else` или оператор `switch`. Затраты машинного времени во многих случаях ничтожно малы. Однако часто это сделать не удается. Оператор `if-else` требует, чтобы обе ветки были успешно вычислены, даже если проверяемое условие на этапе компиляции уже известно. Непонятно? Читайте дальше.

Рассмотрим следующую ситуацию. Нам нужно разработать обобщенный контейнер `NiftyContainer`. Тип его содержимого задается шаблонным параметром.

```
template <class T> class NiftyContainer
{
   ...
};
```

Допустим, что класс `NiftyContainer` содержит указатели на объекты типа `T`. Для дублирования объектов, содержащихся в контейнере, нужно вызвать либо конструктор копирования (для неполиморфных типов), либо виртуальную функцию `Clone()` (для полиморфных типов). Эта информация задается в виде булевского шаблонного параметра.

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
   ...
   void DoSomething()
   {
      T* pSomeObj = ...;
      if (isPolymorphic)
      {
         T* pNewObj = pSomeObj->Clone();
         ... полиморфный алгоритм ...
      }
      else
      {
         T* pNewObj = new T(*pSomeObj);
         ... неполиморфный алгоритм ...
      }
   }
};
```

Проблема заключается в том, что компилятор не считает этот код правильным. Например, поскольку полиморфный алгоритм использует функцию pObj->Clone(), функция NiftyContainer::DoSomething не компилируется, если в классе не определена функция-член Clone(). Правда, уже во время компиляции становится ясно, какая из ветвей оператора if-else выполняется. Однако компилятору эта информация ни к чему — он усердно пытается компилировать обе ветви, даже если оптимизатор впоследствии исключит тупиковую ветвь кода. Если попытаться вызвать функцию DoSomething для класса NiftyContainer<int, false>, компилятор споткнется на вызове функции pObj->Clone() и только презрительно фыркнет в ответ.

Неполиморфная ветвь кода также может не пройти компиляцию. Ошибка происходит, если тип T является полиморфным, а неполиморфная ветвь кода пытается создать новый объект с помощью оператора new T(*pObj). Это может произойти потому, что тип T отключает свой конструктор копирования (объявляя его закрытым), как это положено прилежному полиморфному классу.

Было бы прекрасно, если бы компилятор не анализировал тупиковые ветви, но увы! Итак, что можно сделать в этой ситуации?

Получается, что существует большое количество решений, а класс Int2Type — самое очевидное из них. Он может преобразовывать булевскую переменную isPolymorphic в два разных типа в зависимости от ее значения. Затем можно применить класс Int2Type<Polymorphic> с простой перегрузкой, и готово!

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
private:
   void DoSomething(T* pObj, Int2Type<true>)
   {
       T* pNewObj = pObj->Clone();
       ... полиморфный алгоритм ...
   }
   void DoSomething(T* pObj, Int2Type<false>)
   {
       T* pNewObj = new T(*pObj);
       ... неполиморфный алгоритм ...
   }
```

```
public:
   void DoSomething(T* pObj)
   {
      DoSomething(pObj, Int2Type<isPolymorphic>());
   }
};
```

Класс Int2Type оказывается очень удобным для перевода некоторого значения в тип. После преобразования временную переменную этого типа можно передавать перегруженным функциям, реализующим два требуемых алгоритма.

Этот прием работает благодаря тому, что компилятор не генерирует шаблонные функции, которые никогда не вызываются, — он только проверяет их синтаксис. Таким образом, шаблонный код позволяет осуществлять диспетчеризацию во время компиляции.

В библиотеке Loki класс Int2Type используется в нескольких местах, особенно в главе 11, посвященной мультиметодам. В этой главе шаблонный класс представляет собой механизм двойной диспетчеризации, а шаблонный булевский параметр дает возможность осуществлять симметричную диспетчеризацию.

2.5. Отображение одного типа в другой

В разделе 2.2 указывалось, что частичной специализации шаблонных функций не существует. Иногда, однако, хотелось бы иметь такую возможность. Рассмотрим следующую функцию.

```
template <class T, class U>
T* Create(const U& arg)
{
   return new T(arg);
}
```

Функция Create создает новый объект, передавая аргумент его конструктору.

Предположим, что в нашем приложении принято следующее правило. Объект типа Widget содержит недоступный унаследованный код, и для его создания нужны два аргумента, один из которых представляет собой фиксированное число, скажем, −1. На наш собственный класс, производный от класса Widget, такие ограничения не накладываются.

Каким образом можно специализировать функцию Create так, чтобы она обрабатывала объект класса Widget иначе, чем все другие классы? Очевидное решение таково: нужно создать отдельную функцию CreateWidget. К сожалению, в данный момент не существует универсального интерфейса для создания объектов класса Widget и объектов, производных от этого класса. Это делает функцию Create непригодной для использования в обобщенном коде (generic code).

Функцию невозможно специализировать отдельно, т.е. нельзя написать что-то вроде следующего кода.

```
// Неправильный код — не пытайтесь его применять!
template <class U>
Widget* Create<Widget, U>(const U& arg)
{
   return new Widget(arg, -1);
}
```

В отсутствие частичной специализации функций существует только одно средство — перегрузка. Следовательно, для решения поставленной задачи можно передать функции Create фиктивный объект (dummy object) типа T и применить перегрузку.

```
template <class T, class U>
T* Create(const U& arg, T /* фиктивный объект */)
{
   return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Widget /* фиктивный объект */)
{
   return new Widget(arg, -1);
}
```

Такое решение может повлечь за собой перегрузку конструктора сколь угодно сложного объекта, что также практически бесполезно. Нам необходимо эффективное средство для передачи информации о типе `T` в функцию `Create`. Для этого и нужен класс `Type2Type`. Объект этого класса представляет собой идентификатор, несущий информацию о типе. Его можно передавать перегружаемым функциям.

Рассмотрим определение класса `Type2Type`.

```
template <typename T>
struct Type2Type
{
   typedef T OriginalType;
};
```

Класс `Type2Type` предназначен для передачи любого значения, причем каждому отдельному типу соответствует своя конкретизация, что и требовалось доказать.

Теперь мы можем написать следующий код.

```
// Реализация функции Create на основе перегрузки
// и класса Type2Type
template <Class T, class U>
T* Create(const U& arg, Type2Type<T>)
{
   return new T(arg);
]
template <class U>
Widget* Create(const U& arg, Type2Type<Widget>)
{
   return new Widget(arg, -1);
}
// Используем функцию Create
String* pStr = Create("Hello", Type2Type<String>());
Widget* pW = Create(100, Type2Type<Widget>());
```

Второй параметр функции `Create` предназначен только для выбора подходящей перегрузки. Теперь эту функцию можно специализировать различными конкретизациями класса `Type2Type`, которые преобразовываются в разные типы, используемые в нашем приложении.

2.6. Выбор типа

Иногда в обобщенном коде возникает необходимость выбрать тип в зависимости от булевской константы.

Допустим, что в примере, связанном с классом `NiftyContainer`, который обсуждался в разделе 2.4, в качестве средства для внешнего хранения мы хотим использовать объекты класса `std::vector`. Очевидно, полиморфные типы можно хранить только в виде указателей, а не значений. С другой стороны, мы можем потребовать, чтобы неполиморфные типы сохранялись в виде значений, поскольку это более эффективно.

Рассмотрим шаблонный класс.

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
   ...
};
```

Потребуем, чтобы в нем хранились объекты класса vector<T*> (если значение переменной isPolymorphic равно true) либо объекты класса vector<T> (если значение переменной isPolymorphic равно false). По существу, нам нужен оператор typedef ValueType, значением которого в зависимости от значения булевской переменной isPolymorphic являются типы T* или T.

Для этого можно применить шаблонные классы характеристик (traits) (Alexandrescu, 2000a).

```
template <typename T, bool isPolymorphic>
struct NiftyContainerValueTraits
{
   typedef T* ValueType;
};
template <typename T>
struct NiftyContainerValueTraits<T, false>
{
   typedef T ValueType;
};
template <typename T, bool isPolymorphic>
class NiftyContainer
{
   ...
   typedef NiftyContainerValueTraits<T, isPolymorphic>
      Traits;
   typedef typename Traits::ValueType ValueType;
};
```

Это слишком грубое решение задачи. Более того, оно не масштабируется: для каждого выбираемого типа приходится определять новый шаблонный класс характеристик.

Шаблонный класс Select из библиотеки Loki позволяет выбрать тип прямо на месте. Его определение использует частичную шаблонную специализацию.

```
template <bool flag, typename T, typename U>
struct Select
{
   typedef T Result;
};
template <typename T, typename U>
struct Select<false, T, U>
{
   typedef U Result;
};
```

Если значение переменной flag равно true, компилятор использует первое (обобщенное) определение, и, следовательно, тип Result становится равным T. Если значение переменной flag равно false, вступает в действие специализация, и тип Result становится равным U.

Теперь намного легче определить переменную NiftyContainer::ValueType.

```
template <typename T, bool isPolymorphic>
class NiftyContainer
```

```
{
    ...
    typedef typename Select<isPolymorphic, T*, T>::Result
      ValueType;
    ...
};
```

2.7. Распознавание конвертируемости и наследования на этапе компиляции

При реализации шаблонных функций и классов часто возникает следующий вопрос. Даны два произвольных типа T и U, о которых ничего не известно. Как определить, наследует ли класс U свойства класса T? Распознавание таких отношений является ключевой проблемой при оптимизации обобщенных библиотек. Если класс реализует определенный интерфейс, при работе с обобщенной функцией можно положиться на оптимизированный алгоритм. Такое распознавание на этапе компиляции позволяет избежать динамического приведения типов с помощью оператора dynamic_cast, на что обычно уходит много машинного времени.

Распознавание наследования — более общий процесс, чем распознавание конвертируемости. Общая проблема формулируется следующим образом. Как узнать, возможно ли автоматическое преобразование произвольного типа T в произвольный тип U?

Можно, например, использовать функцию sizeof. Ее мощь удивительна. Эту функцию можно применять к любому сколь угодно сложному выражению, и она вернет размер его результата, не прибегая к вычислениям в ходе выполнения программы. Это означает, что функция sizeof распознает перегрузку, конкретизацию шаблонов, правила преобразования типов — *все*, что относится к выражениям в языке C++. Фактически функция sizeof игнорирует само выражение, возвращая лишь размер его результата.[2]

Идея, на которой основан механизм распознавания конвертируемости, состоит в использовании функции sizeof в сочетании с перегруженными функциями. Для функции предусматриваются две перегрузки. Одна из них получает тип, который должен конвертироваться (в тип U), а вторая — что-нибудь другое (неважно, что именно). Будем вызывать перегруженную функцию с временной переменной типа T, о котором нужно узнать, конвертируется ли он в тип U. Если будет вызвана функция, получающая параметр типа U, значит, тип T конвертируется в тип U. Если будет вызвана нейтральная функция, то тип T не конвертируется в тип U. Для того чтобы определить, какая функция была вызвана, нужно предусмотреть, чтобы две перегруженные функции возвращали значения, типы которых имеют разные размеры. Для определения этих размеров следует вызвать функцию sizeof. Сами по себе типы нас не интересуют — лишь бы они имели разные размеры.

Итак, сначала создадим два типа с разными размерами. (Очевидно, что типы char и long double имеют разные размеры, однако это не гарантируется стандартом.) Безопасная схема выглядит следующим образом.

---

[2] Есть предложение включить в язык C++ оператор typeof, т.е. оператор, возвращающий тип выражения. С его помощью намного проще писать и понимать шаблонные коды. Язык Gnu C++ уже реализовал оператор typeof в качестве своего расширения. Очевидно, что оператор typeof и функция sizeof используют один и тот же код, поскольку в любом случае функция sizeof должна распознавать тип.

```
typedef char Small;
class Big { char dummy[2]; };
```

По определению значение sizeof(Small) равно 1. Размер типа Big неизвестен, но он определенно больше, чем 1 (гарантировать можно только это).

Затем нам понадобятся две перегрузки. Одна из них должна получать параметр типа U, а возвращать — значение типа Small, например:

```
Small Test(U);
```

Как написать функцию, получающую "что-нибудь другое"? Шаблон не подходит, поскольку он всегда квалифицируется как наилучшее соответствие, а значит, скрывает преобразование типов. В данном случае нам нужно "плохое" соответствие, т.е. преобразование, которое происходит, только если автоматическое преобразование невозможно. Бегло просмотрев правила преобразования, применяемые к функции, легко убедиться, что наихудшим является соответствие, порождаемое эллипсисом (ellipsis match), которое находится в самом конце списка.

```
Big Test(...);
```

(Передача объекта в функцию, описание которой содержит эллипсис, может привести к непредсказуемым результатам, но это не имеет значения. На самом деле функция не вызывается. Она даже не реализуется. Напомним, что функция sizeof не вычисляет свой аргумент.)

Теперь нам необходимо применить функцию sizeof к вызову функции Test, передав ей объект типа T.

```
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

Вот и все! При вызове функция Test получает объект, созданный по умолчанию, — T(), а затем функция sizeof вычисляет размер результата этого выражения. Это может быть либо число sizeof(Big), либо число sizeof(Small), в зависимости от того, нашел компилятор преобразование или нет.

Осталась нерешенной одна небольшая проблема. Если объект класса T создается закрытым конструктором, предусмотренным по умолчанию, то выражение T() порождает ошибку во время компиляции, что разрушает всю нашу стройную конструкцию. К счастью, эта проблема имеет простое решение. Достаточно использовать фиктивную функцию, возвращающую объект класса T. (Еще раз напомним, что чудесная функция sizeof на самом деле не вычисляет выражения.) Тогда и компилятор будет сыт, и программа цела.

```
T MakeT(); // Не реализуется
const bool convExists = sizeof(Test(MakeT())) == sizeof(Small);
```

(Кстати, обратите внимание, как остроумно устроена эта проверка — функции MakeT и Test не только не имеют никаких параметров, но и не существуют вообще!)

Итак, все детали этого механизма работают хорошо, и настало время упаковать их в шаблонный класс, который скрывает подробности, связанные с процессом вывода классов, оставляя на поверхности лишь окончательный результат.

```
template <class T, class U>
class Conversion
{
   typedef char Small;
   class Big { char dummy[2]; };
   static Small Test(U);
   static Big Test(...);
```

```
    static T MakeT();
public:
    enum { exists =
        sizeof(Test(MakeT())) == sizeof(Small) };
};
```

Теперь можно испытать шаблонный класс Conversion.

```
int main()
{
    using namespace std;
    cout
      << Conversion<double, int>::exists << ' '
      << Conversion<char, char*>::exists << ' '
      << Convesrion<size_t, vector<int> >::exists << ' ';
}
```

Эта небольшая программа выводит на печать строку “1 0 0”. Обратите внимание на то, что, хотя класс std::vector реализует конструктор, получающий параметр типа size_t, программа возвращает число 0, поскольку это явный конструктор.

В классе можно реализовать еще одну константу Conversion::sameType, принимающую значение true, если классы T и U представляют собой один и тот же тип.

```
template <class T, class U>
class Conversion
{
    ... как и раньше ...
    enum { sameType = false };
};
```

Реализуем константу sameType с помощью частичной специализации шаблонного класса Conversion.

```
template <class T>
class Conversion<T, T>
{
public:
    enum { exists = 1, sameType = 1 };
};
```

Итак, вернемся к исходной задаче. С помощью класса Conversion легко обнаружить наследование.

```
#define SUPERSUBCLASS(T, U) \
    (Conversion<const U*, const T*>::exists && \
    !Conversion<const T*, const void*>::sameType);
```

Макрос SUPERSUBCLASS(T, U) вычисляет значение true, если класс U является производным от класса T, или если классы T и U представляют собой один и тот же тип. Этот макрос выполняет свою работу, распознавая конвертируемость указателей типа const U* в указатели const T*. Возможны только три случая, в которых указатели типа const U* неявно преобразовываются в указатели const T*.

1. Классы T и U представляют собой один и тот же тип.
2. Класс T является единственным открытым базовым классом для класса U.
3. Класс T представляет собой тип void.

Последнее исключается при второй проверке. На практике бывает полезно считать первый вариант (классы T и U представляют собой один и тот же тип) вырожденным случаем отношения “является” (“is-a”), поскольку с практической точки зрения лю-

бой класс часто можно считать собственным суперклассом. Если проверку нужно ужесточить, можно написать следующий код.

```
#define SUPERSUBCLASS_STRICT(T, U) \
   (SUPERSUBCLASS(T, U) && \
   !(Conversion<const T, const U>::sameType)
```

Зачем в этом фрагменте кода столько модификаторов const? Причина состоит в том, что проверка не должна завершаться неудачей из-за проблем, связанных с этими модификаторами. Если шаблонный код применяет модификатор const дважды (к типу, который уже является константным), второй модификатор игнорируется. Короче говоря, модификатор const в макросе SUPERSUBCLASS применяется в целях безопасности.

Почему макрос назван SUPERSUBCLASS, а не BASE_OF или INHERITS? По одной очень важной причине. Первоначально в библиотеке Loki этот макрос назывался INHERITS. Однако при использовании выражения INHERITS(T, U) каждый раз возникал вопрос, что именно проверяется: то, что класс T является производным от класса U, или наоборот? Очевидно, что выражение SUPERSUBCLASS(T, U) таких сомнений не вызывает, поскольку в его названии первая часть (SUPER) относится к первому параметру T, а вторая (SUB) — ко второму параметру U.

2.8. Оболочка вокруг класса type_info

В стандарте языка C++ предусмотрен класс std::type_info, позволяющий исследовать типы объектов в ходе выполнения программы. Обычно класс type_info применяется в сочетании с оператором typeid, возвращающим ссылку на объект класса type_info.

```
void Fun(Base* pObj)
{
   // Сравнивает два объекта типа type_info с типами
   // *pObj и Derived, соответственно
   if (typeid(*pObj) == typeid(Derived)
   {
      ... ага, на самом-то деле указатель pObj
      ссылается на объект класса Derived ...
   }
   ...
}
```

Оператор typeid возвращает ссылку на объект класса type_info. Кроме операторов сравнения operator== и operator!=, класс type_info содержит еще две функции.

- Функция-член name возвращает текстуальное представление типа в форме переменной типа const char*. Стандартного способа преобразовывать имена классов в строки нет. Поэтому не следует ожидать, что значением выражения typeid(Widget) будет строка “Widget”. Вполне приемлемо (хотя и не слишком хорошо), если реализация функции-члена type_info::name возвратит для всех типов пустую строку.
- Функция-член before устанавливает между объектами типа type_info отношение порядка. Используя эту функцию, можно индексировать объекты класса type_info.

К сожалению, полезные свойства класса type_info реализованы так, что их трудно эксплуатировать. В классе type_info не предусмотрены конструктор копирования и оператор присваивания, что не позволяет хранить в памяти объекты этого типа. Однако можно хранить *указатели* на объекты типа type_info. Объекты, возвращаемые оператором

typeid, хранятся в статической памяти, поэтому беспокоиться о времени их жизни не стоит. Вместо этого следует обеспечить *идентичность указателей* (pointer identity).

Стандарт языка C++ не гарантирует, что при каждом вызове, например, оператора typeid(int), возвращается ссылка на один и тот же объект класса type_info. Следовательно, сравнить указатели на объекты класса type_info невозможно. Хранить указатели на объекты этого типа и сравнивать их между собой нужно с помощью функции type_info::operator==, которая применяется к разыменованным указателям.

При необходимости рассортировать объекты класса type_info снова нужно сохранить указатели на них, но на этот раз использовать функцию-член before. Следовательно, для того чтобы применить упорядоченные контейнеры из библиотеки шаблонов STL, нужно написать небольшой функтор (functor) и поработать с указателями.

Все это довольно неудобно. Для того чтобы преодолеть эти трудности, создадим вокруг класса type_info интерфейсный класс (wrapper class), в котором хранится указатель на объект типа type_info и предусмотрено следующее.

- Все функции-члены класса type_info.
- Семантика значений (открытый конструктор копирования и оператор присваивания).
- Операторы сравнения operator< и operator==.

В библиотеке Loki определен интерфейсный класс TypeInfo, в котором реализована описанная выше оболочка класса type_info. Вот его краткий обзор.

```
class TypeInfo
{
public:
   // Конструкторы/деструкторы
   TypeInfo(); // Необходим для контейнеров
   TypeInfo(const std::type_info&);
   TypeInfo(const TypeInfo&);
   TypeInfo& operator==(const TypeInfo&);
   // Функции сравнения
   bool before(const TypeInfo&) const;
   const char* name() const;
private:
   const std::type_info* pInfo_;
}
// Операторы сравнения
   bool operator==(const TypeInfo&, const TypeInfo&);
   bool operator!=(const TypeInfo&, const TypeInfo&);
   bool operator<(const TypeInfo&, const TypeInfo&);
   bool operator<=(const TypeInfo&, const TypeInfo&);
   bool operator>(const TypeInfo&, const TypeInfo&);
   bool operator>=(const TypeInfo&, const TypeInfo&);
```

Благодаря конструктору преобразования (conversion constructor), получающему в качестве параметра объект класса std::type_info, можно непосредственно сравнивать объекты типов TypeInfo и std::type_info, как показано ниже.

```
void Fun(Base* pObj)
{
   TypeInfo info = typeid(Derived);
   ...
   if (typeid(*pObj) == info)
   {
      ... Указатель pBase действительно указывает
```

```
            на объект класса Derived ...
    }
    ...
}
```

Способность копировать и сравнивать между собой объекты класса TypeInfo важна во многих ситуациях. Фабрики клонирования, описанные в главе 8, и механизм двойной диспетчеризации, рассмотренный в главе 11, достаточно ярко иллюстрируют этот факт.

2.9. Классы NullType и EmptyType

В библиотеке Loki определены два очень простых типа: NullType и EmptyType. Их можно использовать для идентификации более широких типов.

Класс NullType служит в качестве нулевого маркера типов (null marker).

```
class NullType();
```

Обычно объекты этого класса не создаются. Его единственное предназначение — обозначить типы, не представляющие интереса. В разделе 2.10 класс NullType используется в ситуациях, когда тип имеет синтаксический смысл, но не имеет семантического. (Например: "На объекты какого типа указывает переменная типа int?".) Кроме того, списки типов, описанные в главе 3, используют класс NullType в качестве маркера конца списка и для возврата сообщения "тип не найден".

Второй вспомогательный тип — класс EmptyType. Как и следовало ожидать, его определение имеет следующий вид.

```
struct EmptyType {};
```

Этот тип можно использовать в качестве базового класса, а также для передачи значений типа EmptyType. Кроме того, его можно применять в шаблонах в качестве типа, заданного по умолчанию ("не важно какой").

2.10. Характеристики типов

Характеристики (traits) — это метод обобщенного программирования, позволяющий принимать решения на этапе компиляции программы, основываясь на информации о типах, аналогично тому, как в ходе выполнения программы принимаются решения, основанные на значениях (Alexandrescu, 2000a). Предоставляя "дополнительный окольный путь", позволяющий решить многие проблемы, связанные с проектированием программного обеспечения, характеристики дают возможность принимать решения, основываясь на информации о типах, находясь вне конкретного контекста. Код, полученный в результате, становится яснее, читабельнее и легче в эксплуатации.

Обычно программисты пишут свои собственные характеристические шаблоны и классы для обобщенного кода. Однако характеристики можно применять к любым типам. Они помогают программисту, создающему обобщенный код, точнее согласовать шаблонный код с возможностями типа.

Допустим, что мы реализуем алгоритм копирования.

```
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
   for (; first != last; ++first, ++result)
      *result = *first;
}
```

Теоретически такой алгоритм реализовывать не имеет смысла, поскольку он дублирует возможности библиотечной функции std::copy. Однако иногда возникает необходимость специально настроить функцию копирования на определенный тип.

Предположим, что мы разрабатываем код для многопроцессорной машины, в которой предусмотрена очень быстрая элементарная функция BitBlast. Естественно, мы хотели бы использовать это преимущество там, где это возможно.

```
// Прототип функции BigBlast в заголовочном файле
// "SIMD_Primitives.h"
void BigBlast(const void* src, void* dest, size_t bytes);
```

Разумеется, функция BigBlast предназначена для копирования только элементарных типов и простых старых структур данных. Эту функцию нельзя применять к типам, имеющим нетривиальный конструктор копирования. Таким образом, желательно было бы реализовать новую функцию Copy так, чтобы использовать преимущества функции BigBlast с максимальной выгодой и при копировании объектов более сложных типов. При копировании элементарных типов функция Copy "совершенно непонятным образом" будет выполняться быстрее.

Для того чтобы реализовать функцию Copy, нужно выполнить две проверки.

- Являются ли переменные InIt и OutIt обычными указателями (в отличие от более сложных типов итераторов)?
- Можно ли копировать объекты, на которые ссылаются указатели InIt и OutIt, побитово?

Если на этапе компиляции мы положительно ответим на оба вопроса, можно применять функцию BigBlast. В противном случае нужно использовать обобщенный цикл for.

Решить такие проблемы можно с помощью характеристик. В этой главе, в основном, используется реализация характеристик из библиотеки Boost C++ (Boost).

2.10.1. Реализация характеристик указателей

В библиотеке Loki определен шаблонный класс TypeTraits, в котором собрано множество характеристик обобщенных типов. Этот класс специализирует шаблоны, содержащиеся в нем, и представляет результаты.

Реализация характеристик большинства типов основывается на полной или частичной специализации шаблонов (раздел 2.2). Например, приведенный ниже фрагмент кода определяет, является ли тип T указателем.

```
template <typename T>
class TypeTraits
{
private:
   template <class U> struct PointerTraits
   {
      enum(result = false };
      typedef NullType PointeeType;
   };
   template <class U> struct PointerTraits<U*>
   {
      enum { result = true };
      typedef U PointeeType;
   };
public:
   enum { isPointer = PointerTraits<T>::result };
```

```
    typedef PointerTraits<T>::PointeeType PointeeType;
    ...
};
```

В первом определении вводится шаблонный класс PointerTraits, который как бы говорит: "Класс T — не указатель". Напомним, что ранее (в разделе 2.9) в этих ситуациях применялся класс NullType.

Во втором определении (выделенном в тексте) вводится частичная специализация шаблонного класса PointerTraits, соответствующая любому типу указателей. Для нулевого указателя, который не ссылается ни на один объект, специализация, выделенная в тексте фрагмента, квалифицируется как соответствие, более точное, чем обобщенный шаблон любого типа указателей. Следовательно, вступает в силу специализация для указателей, а переменная result принимает значение true. Класс PointeeType определяется соответствующим образом.

Теперь можно понять внутреннее устройство реализации класса std::vector::iterator — является он простым указателем или представляет собой некий сложный тип?

```
int main()
{
    const bool
        iterIsPtr = TypeTraits<vector<int>::iterator>::isPointer;
    cout << "vector<int>::iterator is " <<
        iterIsPtr ? "fast" : "smart" << '\n';
}
```

Аналогично в классе TypeTraits реализуется константа isReference и определяется тип ReferenceType. Для ссылочного типа T класс ReferenceType представляет собой тип, на который ссылается объект класса T. Если класс T является обычным типом, то класс ReferencedType совпадает с ним.

Разпознавание указателей на члены класса (глава 5) немного отличается от описанного выше. Для этого нужна следующая специализация.

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PToMTraits
    {
        enum { result = false };
    };
    template <class U, class V>
    struct PToMTraits<U V::*>
    {
        enum { result = true };
    };
public:
    enum { isMemberPointer = PToMTraits<T>::result };
    ...
};
```

2.10.2. Распознавание основных типов

Класс TypeTraits<T> реализует статическую константу isStdFundamental. По значению этой константы можно определить, является класс T стандартным основным типом или нет. К этим типам относится тип void и все числовые типы (которые,

в свою очередь, подразделяются на типы чисел с плавающей точкой и целочисленные типы). В классе `TypeTraits` реализуются константы, показывающие категорию, к которой может принадлежать заданный тип.

Немного забегая вперед, следует сказать, что *списки типов* (глава 3) позволяют легко определять, принадлежит ли тип заданному множеству типов. Пока нам нужно знать лишь то, что приведенное ниже выражение (в котором суффикс `nn` означает количество типов, перечисленных в списке) возвращает позицию класса `T` в списке, начиная отсчет от нуля, или число −1, если заданного класса в списке нет.

```
TL::IndexOf<T, TYPELIST_nn(список типов, разделенных запятыми)>::value
```

Например, значение выражения

```
TL::IndexOf<T, TYPELIST_4(signed char, short int,
   int, long int)>::value
```

больше или равно нулю, только если класс `T` является целочисленным типом со знаком.

Ниже приведен фрагмент определения класса `TypeTrait` для основных типов.

```
template <typename T>
class TypeTraits
{
   ... как показано выше ...
public:
   typedef TYPELIST_4(
      unsigned char, unsigned short int,
      unsigned int, unsigned long int)
   UnsignedInts;
   typedef TYPELIST_4(signed char, short int, int, long int)
   SignedInts;
   typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
   typedef TYPELIST_3(float, double, long double) Floats;
   enum { isStdUnsignedInt =
      TL::IndexOf<T, UnsignedInts>::value >= 0 };
   enum { isStdSignedInt =
      TL::IndexOf<T, SignedInts>::value >= 0 };
   enum { isStdIntegral = isStdUnsignedInt || isStdSignedInt ||
      TL::IndexOf<T, OtherInts>::value >= 0 };
   enum { isStdFloat = TL::IndexOf<T, Floats>::value >= 0 }
   enum { isStdArith = isStdIntegral || isStdFloat };
   enum { isStdFundamental = isStdArith || isStdFloat ||
      Conversion<T, void>::sameType };
   ...
};
```

Использование списков и класса `TL::IndexOf` дает возможность быстро получать информацию о типах, не прибегая к многократной специализации шаблонов. Если вы не можете устоять перед соблазном покопаться в деталях, можете сразу перейти к главе 3, но не забудьте вернуться обратно.

Фактическая реализация процедуры распознавания основных типов намного сложнее, что позволяет применять ее для распознавания более широкого спектра классов, разрабатываемых производителями программного обеспечения (например, `int64` или `long long`).

2.10.3. Оптимальные типы параметров

В шаблонном коде иногда нужно ответить на следующий вопрос. Задан произвольный тип `T`. Какой способ передачи и получения объектов типа `T` в качестве аргу-

ментов функций наиболее эффективен? В общем случае сложные типы эффективнее всего передавать по ссылке, а скалярные — по значению. (К скалярным типам относятся арифметические типы, описанные ранее, например enum, указатели и указатели на члены класса.) Для сложных типов это позволяет избежать ненужных затрат времени на вызовы конструкторов и деструкторов, а для скалярных типов — избыточных способов доступа к переменным, предоставляемых ссылками.

Не забудьте, что в языке C++ указатели на ссылки не допускаются. Таким образом, если объект класса T уже является ссылкой, то новую ссылку на нее создавать не следует.

Несложный анализ оптимальных типов параметров функции приводит к следующему алгоритму. Рассмотрим тип параметра с именем ParameterType.

Если класс T — это ссылка на некоторый тип, классы ParameterType и T совпадают (ссылки на ссылки не допускаются).
В противном случае,
  если класс T — это скалярный тип (int, float и т.п.), то класс ParameterType — это класс T (основные типы лучше всего передавать по значению),
  иначе класс ParameterType является типом const T& (неэлементарные типы лучше передавать по ссылке).

Одно из достоинств этого алгоритма состоит в том, что он позволяет избежать ошибки, связанной с использованием ссылки на ссылку, которая может возникнуть при совместном использовании функций bind2nd и mem_fun из стандартной библиотеки.

Класс TypeTraits легко реализовать с помощью приема, который мы уже применяли, и определенных выше характеристик ReferencedType и isPrimitive.

```
template <typename T>
class TypeTraits
{
   ... как и раньше ...
public:
   typedef Select<isStdArith || isPointer || isMemberPointer,
      T, ReferencedType&>::Result
       ParameterType;
};
```

К сожалению, эта схема не позволяет передавать по значению параметры перечислимых типов, поскольку невозможно определить, является заданный тип перечислимым или нет.

Класс TypeTraits::ParameterType используется в шаблонном классе Functor, определенном в главе 5.

2.10.4. Удаление квалификаторов

Имея тип T, можно легко получить константу, очень похожую на обычную константу const T. Однако выполнить обратную операцию (т.е. удалить квалификатор const) намного труднее. Кроме того, иногда возникает необходимость избавиться и от квалификатора типа volatile.

Рассмотрим, например, создание интеллектуального указателя SmartPtr (глава 7). Если бы мы захотели предоставить пользователю возможность создавать интеллектуальные указатели на константные объекты, например SmartPtr<const Widget>, пришлось бы модифицировать указатель на объект класса Widget. В этом случае в классе SmartPtr следовало бы создавать объект класса Widget на основе константного объекта.

Реализовать такой “ликвидатор” достаточно просто. Для этого нужно применить частичную шаблонную специализацию.

```
template <typename T>
class TypeTraits
{
   ... как и раньше ...
private:
   template <class U> struct UnConst
   {
      typedef U Result;
   };
   template <class U> struct UnConst<const U>
   {
      typedef U Result;
   }
public:
     typedef UnConst<T>Result NonConstType;
};
```

2.10.5. Применение класса TypeTraits

Класс TypeTraits предоставляет много интересных возможностей. Например, с помощью описанных выше приемов можно реализовать процедуру Copy, использующую функцию BitBlast (проблема, упомянутая в разделе 2.10). Для этого можно применить класс TraitsType, позволяющий получить информацию о двух итераторах, и шаблонный класс Int2Type, осуществляющий диспетчеризацию вызова либо функции BitBlast, либо классической процедуры копирования.

```
enum CopyAlgoSelector { Conservative, Fast };

// Классическая процедура применяется ко всем типам
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result,
   Int2Type<Conservative>
{
   for (; first != last; ++first, ++result)
      *result = *first;
      return result;
}
// Быстрая процедура применяется только
// для указателей на простые данные
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIn first, InIt last, OutIt result,
   Int2Type<Fast>)
{
   const size_t n = last-first;
   BitBlast(first, result, n * sizeof(*first));
   return result + n;
}
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
   typedef TypeTraits<InIt>::PointeeType SrcPointee;
   typedef TypeTraits<OutIt>::PointeeType DestPointee;
   enum { copyAlgo =
      TypeTraits<InIt>::isPointer &&
      TypeTraits<OutIt>::isPointer &&
      TypeTraits<SrcPointee>::isStdFundamental &&
```

```
        TypeTraits<DestPointee>::isStdFundamental &&
        sizeof(SrcPointee) == sizeof(DestPointee) ? Fast :
            Conservative };
        return CopyImpl(first, last, result, Int2Type<copyAlgo>);
}
```

Несмотря на то что процедура Copy сама по себе не слишком сложна, в ней есть один интересный момент. Перечислимое значение copyAlgo позволяет делать выбор среди разных реализаций. Логика этого выбора такова: функция BitBlast используется, если оба итератора являются указателями, и оба типа, на которые ссылаются указатели, являются основными и имеют одинаковый размер. Последнее условие необычно. Рассмотрим такой фрагмент кода.

```
int* p1 = …;
int* p2 = …;
unsigned int* p3 = …;
Copy(p1, p2, p3);
```

В этом варианте функция Copy вызовет функцию быстрого копирования, как и положено, хотя типы источника и адресата отличаются друг от друга.

Недостаток функции Copy заключается в том, что она не ускоряет то, что можно было бы ускорить. Например, для простой структуры, характерной для языка C, содержащей лишь данные элементарных типов — так называемой структуры *старых простых данных* (old plain data structure), или POD-структуры, стандартом предусмотрено побитовое копирование. Однако функция Copy не распознает простые структуры и вызывает медленную процедуру копирования. Здесь, кроме класса TypeTraits, нужно снова применить классические характеристики.

```
template <typename T> struct SupportBitwiseCopy
{
   enum { result = TypeTraits<T>::isStdFundamental };
};
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result,
   Int2Type<true>)
{
   typedef TypeTraits<InIt>::PointeeType SrcPointee;
   typedef TypeTraits<OutIt>::PointeeType DestPointee;
  enum { useBitBlast =
      typeTraits<InIt>::isPointer &&
      typeTraits<OutIt>::isPointer &&
      SupportBitwiseCopy<SrcPointee>::result &&
          SupportBitwiseCopy<DestPointee>::result &&
          sizeof(SrcPointee) == sizeof(DestPointee) }
   return CopyImpl(first, last, Int2Type<useBitBlast>);
}
```

Теперь, чтобы применить функцию BitBlast для POD-типов, достаточно специализировать шаблонный класс SupportBitwiseCopy и задать в нем значение true.

```
template<> struct SupportBitwiseCopy<MyCopy>
{
   enum { result = true };
};
```

2.10.6. Заключение

В табл. 2.1 показано все множество характеристик, реализованных в библиотеке Loki.

2.11. Резюме

Описанные приемы программирования образуют строительные конструкции для создания компонентов, представленных в книге. Большинство этих приемов основано на шаблонных кодах.

- *Статические диагностические утверждения* (раздел 2.1) позволяют библиотекам генерировать осмысленные сообщения об ошибках.
- *Частичная шаблонная специализация* (раздел 2.2) позволяет специализировать шаблоны не для фиксированного специального набора параметров, а для семейства параметров, наиболее соответствующих заданному шаблону.
- *Локальные классы* (раздел 2.3) допускают интересные возможности, особенно в отношении шаблонных функций.
- *Отображение целочисленных констант в типы* (раздел 2.4) облегчает процесс статической диспетчеризации, основанной на числовых значениях (особенно булевского типа).
- *Отображение между типами* (раздел 2.5) позволяет заменить перегрузку функций частичной специализацией шаблонов (свойство, которого в стандарте языка C++ нет).
- *Выбор типов* (раздел 2.6) позволяет выбирать типы на основе анализа условных выражений.
- *Распознавание конвертируемости и наследования на этапе компиляции* (раздел 2.7) дает возможность получать информацию о том, можно ли преобразовать два заданных типа друг в друга, являются они псевдонимами одного и того же типа или наследуют свойства друг друга.
- *Класс* `TypeInfo` (раздел 2.8) реализует оболочку для класса `std::type_info`, определяя семантику значений и операции сравнения.
- Классы `NullType` и `EmptyType` (раздел 2.9) играют роль структурного нуля в метапрограммировании шаблонов.
- Шаблон `TypeTraits` (раздел 2.10) предоставляет множество характеристик общего предназначения, которые можно использовать для настройки кода на специфическую категорию типов.

Таблица 2.1. Члены класса TypeTraits<T>

Имя	Вид	Описание
`isPointer`	Булевская константа	Принимает значение `true`, если класс `T` является параметром
`PointeeType`	Тип	Является типом, на который ссылается тип `T`, если класс `T` является указателем, в противном случае является типом `NullType`
`isReference`	Булевская константа	Принимает значение `true`, если класс `T` является ссылочным типом
`ReferencedType`	Тип	Если класс `T` является ссылочным типом, класс `ReferencedType` является типом, на который ссылается класс `T`. В противном случае класс `ReferencedType` сам является классом `T`

Окончание табл. 2.1

Имя	Вид	Описание
ParameterType	Тип	Оптимально соответствует параметру неизменяемой функции. Может быть либо типом T, либо типом const T&
isConst	Булевская константа	Принимает значение true, если класс T является константным типом
NonConstType	Тип	Удаляет квалификатор const у типа T, если он у него есть
isVolatile	Булевская константа	Принимает значение true, если класс T является типом с квалификатором volatile
NonVolatileType	Тип	Удаляет квалификатор volatile у типа T, если он у него есть
NonQualifiedType	Тип	Удаляет квалификаторы const и volatile у типа T, если они у него есть
isStdUnsignedInt	Булевская константа	Принимает значение true, если класс T является одним из следующих четырех беззнаковых типов (unsigned char, unsigned short int, unsigned int или unsigned long int)
isStdSignedInt	Булевская константа	Принимает значение true, если класс T является одним из следующих четырех типов со знаком (signed char, short int, signed int или long int)
isStdIntegral	Булевская константа	Принимает значение true, если класс T является обычным целочисленным типом
isStdFloat	Булевская константа	Принимает значение true, если класс T является обычным типом чисел с плавающей точкой (float, double или long double)
isStdArith	Булевская константа	Принимает значение true, если класс T является стандартным арифметическим типом (целочисленным или с плавающей точкой)
isStdFundamental	Булевская константа	Принимает значение true, если класс T является основным типом (арифметическим или void)

3

Списки типов

Списки типов представляют собой средство для манипулирования коллекциями типов. При этом к типам можно применять все основные операции, предусмотренные для обычных списков.

В некоторых шаблонах проектирования описываются коллекции типов, над которыми производятся манипуляции. Эти типы либо связаны между собой отношением наследования, либо не зависят друг от друга. Яркими примерами таких шаблонов проектирования являются шаблоны `Abstract Factory` и `Visitor` (Gamma et al., 1995). Используя традиционные методы программирования, можно манипулировать коллекциями типов с помощью явного повторения. Это приводит к раздуванию кода. Многие люди считают, что ничего лучше придумать нельзя. Однако списки типов позволяют автоматизировать задачи, которые программисты обычно решают с помощью макросов редактора. Списки типов придают языку C++ необычайную мощь, позволяя создавать новые, интересные идиомы.

В этой главе содержится полное описание возможностей списков типов в языке C++, а также приводятся многочисленные примеры их применения. Здесь представлена следующая информация.

- Концепция списков типов.
- Способы создания и функционирования списков типов.
- Методы эффективного манипулирования списками типов.
- Применение списков типов и программирование идиом, поддерживаемых списками типов.

В главах 9–11 показано, как списки типов используются в качестве технологии программирования.

3.1. Зачем нужны списки типов

Иногда нужно написать один и тот же код для большого количества разных типов, и шаблоны тут ничем не могут помочь. Рассмотрим, например, реализацию шаблона `Abstract Factory` (Gamma et al., 1995). Ниже приведено определение виртуальной функции для каждого типа из некоторого набора, заданного на этапе проектирования.

```
class WidgetFactory
{
public:
   virtual Window* CreateWindow() = 0;
   virtual Button* CreateButton() = 0;
   virtual ScrollBar* CreateScrollBar() = 0;
};
```

Если концепцию абстрактной фабрики необходимо обобщить и реализовать в виде библиотеки, нужно предоставить пользователю возможность создавать фабрики произвольных наборов типов, а не только типов Window, Button и ScrollBar. Шаблоны этой способностью не обладают.

Хотя на первый взгляд шаблон Abstract Factory не предоставляет широких возможностей для абстрагирования и обобщения, некоторые моменты достойны более глубокого изучения.

1. Если вы не можете обобщить основную концепцию, то вряд ли вам удастся обобщить ее конкретный пример. Это очень важный принцип. Если некая сущность не допускает обобщения, то программист вынужден постоянно иметь дело с конкретными воплощениями этой сущности. Например, несмотря на то, что базовый класс абстрактной фабрики довольно прост, при реализации различных конкретных фабрик приходится прибегать к многократному повторению одного и того же кода.
2. Функциями-членами класса WidgetFactory невозможно свободно манипулировать (см. приведенный выше код). Набор сигнатур виртуальных функций в принципе невозможно сделать обобщенным. В качестве примера рассмотрим следующий код.

   ```
   template <class T>
   T* MakeRedWidget(WidgetFactory& factory)
   {
       T* pW = factory.CreateT(); // Невозможно!!!
       pW->SetColor(RED);
       return pW;
   }
   ```

   Нужно вызвать функции CreateWindow, CreateButton или CreateScrollBar, в зависимости от того, что представляет собой класс T — класс Window, Button или ScrollBar соответственно. В языке C++ невозможно добиться такой подстановки текста.
3. И последнее (по порядку, но не по важности) — библиотеки только выиграли от того, что прекратились бесконечные дискуссии о соглашениях, принимаемых для записи имен (createWindow, create_window или CreateWindow), и тому подобных мелочах. В библиотеках реализован легкий и стандартный способ принятия подобных решений. Абстрактные фабрики также должны были бы иметь такую возможность.

Итак, подытожим наши пожелания. Что касается п. 1, было бы прекрасно, если бы мы смогли создавать объекты класса WidgetFactory, передавая список параметров шаблонному классу AbtsractFactory.

```
typedef AbstractFactory<Window, Button, ScrollBar> WidgetFactory;
```

Из п. 2 следует, что нам необходим шаблонный вызов для разных функций вида Create*Xxx*, например, Create<Window>(), Create<Button>() и т.д. Тогда мы смогли бы вызывать эти функции с помощью обобщенного кода.

```
template <class T>
T* MakeRedWidget(WidgetFactory& factory)
{
   T* pW = Factory.Create<T>(); // Вот это прекрасно!
   pW->SetColor(RED);
   return pW;
}
```

Однако мы не можем реализовать эти пожелания. Во-первых, оператор typedef для класса WidgetFactory применить невозможно, поскольку шаблоны не могут иметь переменное количество параметров. Во-вторых, шаблонный синтаксис Create*Xxx*() недопустим, так как виртуальные функции не могут быть шаблонными.

Итак, очевидно, что правила языка программирования не позволяют достичь высокого уровня абстракции и возможностей повторного использования кода.

Списки типов позволяют снять эти ограничения и создать не только обобщенные абстрактные фабрики, но и реализовать много других шаблонов проектирования.

3.2. Определение списков типов

По разным причинам язык C++ иногда позволяет программисту сказать: "Это лучшие пять строк кода, которые я когда-либо написал!". Возможно, это связано с семантическим богатством этого языка или с необычностью его свойств. По этой традиции списки типов выглядят крайне просто.

```
template <class T, class U>
struct Typelist
{
   typedef T head;
   typedef U Tail;
};
namespace TL
{
   ... алгоритмы работы со списками типов ...
}
```

Все, что относится к спискам типов, за исключением определения самого класса Typelist, пребывает в пространстве имен TL. В свою очередь, пространство имен TL находится внутри пространства имен библиотеки Loki, как и весь код этой библиотеки. Для того чтобы упростить примеры, в этой главе упоминания пространства имен TL пропущены. Читателю следует помнить это, используя заголовочный файл Typelist.h. (Если вы забудете, компилятор вам напомнит!)

Класс Typelist содержит два типа. Доступ к ним обеспечивается внутренними именами Head и Tail. Вот именно! Нам не нужны списки типов, содержащие три и больше элементов, поскольку они у нас уже есть. Например, рассмотрим список типов, состоящий из трех вариантов типа char.

```
typedef Typelist<char, Typelist<signed char, unsigned char> >
   CharList;
```

(Обратите внимание на раздражающий, но необходимый пробел между двумя грамматическими лексемами (token) > в конце строки.)

Списки типов не содержат никаких значений. Их тела пусты, они не имеют никакого состояния и не определяют никаких функциональных возможностей. Во время выполнения программы списки типов не содержат вообще никаких значений. Их единственное предназначение — предоставлять информацию о типах. Следовательно, любая обработка списков типов возможна лишь на этапе компиляции, а не в ходе выполнения программы. Списки типов не предназначены для создания объектов, хотя в их создании нет ничего опасного. Таким образом, термин "список типов" означает тип списка, а не его значение. Значения списков типов не представляют никакого интереса. На практике применяются только их типы. (В разделе 3.13.2 показано, как списки типов можно использовать для создания коллекций значений.)

Здесь используется одна особенность, состоящая в том, что шаблонный параметр может иметь любой тип и быть конкретизацией своего собственного шаблона. Это старое, хорошо известное свойство шаблонов, которое часто применяется при создании таких необычных матриц, как `vector < vector<double> >`. Поскольку класс `Typelist` имеет два параметра, его всегда можно расширить, заменив один из параметров другим классом `Typelist`, и так до бесконечности.

Однако существует одна небольшая проблема. Пока мы можем создавать списки, состоящие из нескольких типов, но у нас нет инструмента для описания списков, не содержащих никаких типов или содержащих только один тип. Для этого необходим *нулевой тип списка* (null list type) и класс `NullType`, описанный в главе 2, предназначен именно для этого.

Примем соглашение, что каждый список типов должен заканчиваться классом `NullType`, служащим маркером конца списка. Это намного удобнее традиционного нулевого байта "\0", который применяется для обозначения конца строк в традиционных функциях языка С. Теперь можно дать определение класса `Typelist`, состоящего лишь из одного элемента.

```
// Определение класса NullType дано в главе 2
typedef Typelist<int, NullType> OneTypeOnly;
```

Определение класса `Typelist`, состоящего из трех типов `char`, принимает следующий вид.

```
typedef Typelist<char, Typelist<signed char,
   Typelist<unsigned char, NullType> > > AllCharTypes;
```

Следовательно, мы получили шаблон неограниченного списка типов `Typelist`, который может содержать любое их количество.

Посмотрим теперь, как можно манипулировать списками типов. (Как и прежде, это относится к типам `Typelist`, а не к объектам типа `Typelist`.) Приготовьтесь к приключениям. С этого момента мы погружаемся в подземелье языка C++, мир странных новых правил — мир программирования на этапе компиляции.

3.3. Линеаризация создания списков типов

Сами по себе списки типов слишком напоминают конструкцию из языка LISP, поэтому их нелегко использовать. Такие конструкции очень нравятся программистам на языке LISP, но они не очень хорошо согласуются с языком C++ (не говоря уже о пробелах между символами <, которые нельзя забывать). Например, вот как выглядит список целочисленных типов.

```
typedef Typelist<signed char,
   Typelist<short int,
     Typelist<int, Typelist<long int, NullType> > > >
   SignedIntegrals;
```

Списки типов были бы превосходной концепцией, но они явно нуждаются в более привлекательной упаковке.

Для того чтобы упростить процедуру создания списков типов, в файле `Typelist.h` из библиотеки Loki определено большое количество макросов, преобразующих рекурсию в простое перечисление, правда, за счет утомительных повторений. Однако это — не проблема. Повторение выполняется только один раз, в библиотечном коде. Это позволяет масштабировать списки типов для большого количества элементов (50). Типичный макрос выглядит следующим образом.

```
#define TYPELIST_1(T1) Typelist<T1, NullType>
#define TYPELIST_2(T1, T2) Typelist<T1, TYPELIST_1(T2) >
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3) >
#define TYPELIST_4(T1, T2, T3, T4) \
   Typelist<T1, Typelist_3(T2, T3, T4)
...
#define TYPELIST_50(...) ...
```

Каждый макрос использует предыдущий, что позволяет пользователю библиотеки при необходимости легко увеличивать верхний предел.

Теперь можно сформулировать более удобное определение списка целочисленных типов SignedIntegrals.

```
typedef TYPELIST_4(signed char, short int, long int)
   SignedIntegrals;
```

Линеаризация создания списков типов — всего лишь начало. Манипуляции со списками типов все еще неудобны. Например, доступ к последнему элементу списка SignedIntegral вынуждает использовать конструкцию SignedIntegrals::Tail::Tail::Head. Не вполне понятно, как мы сможем манипулировать списками типов в обобщенном виде. Итак, пришло время определить основные операции над списками типов в терминах элементарных операций над списками значений.

3.4. Вычисление длины списка

Рассмотрим простую операцию. Задан список типов TList. На этапе компиляции нужно получить константу, равную его длине. Эта константа должна быть статической, поскольку список типов является статической конструкцией, и естественно ожидать, что все вычисления, относящиеся к нему, выполняются именно на этапе компиляции.

Идея, лежащая в основе большинства операций над списками типов, заключается в применении рекурсивных шаблонов, использующих для своего определения собственные конкретизации. При этом такие шаблоны передают разные списки шаблонных аргументов. Рекурсия, полученная таким способом, заканчивается явной специализацией предельного варианта (border case).

Код, вычисляющий длину списка типов, довольно лаконичен.

```
template <class TList> struct Length;
template <> struct Length<NullType>
{
   enum { value = 0 };
};
template <class T, class U>
struct Length< Typelist<T, U> >
{
   enum { value = 1 +Length<U>::value };
};
```

В переводе на человеческий язык, это означает: "Длина нулевого списка типов равна 0. Длина любого другого списка типов равна 1 плюс длина его оставшейся части".

Реализация структуры Length использует *частичную шаблонную специализацию* (глава 2), позволяющую различать нулевой тип и список типов. Первая специализация структуры Length является полной и соответствует только типу NullType. Вторая, частичная, специализация соответствует любому классу Typelist<T, U>, включая составные списки, в которых класс U, в свою очередь, является классом Typelist<V, W>.

Во второй специализации вычисления проводятся рекурсивно. Величина `value` в ней определяется как 1 (с учетом головы списка `T`) плюс длина хвоста списка. Когда в хвосте списка остается единственный класс `NullType`, обнаруживается совпадение с первым определением, и рекурсия останавливается. В результате вычисляется величина, равная длине списка. Допустим, например, что нам нужно определить массив в стиле языка С, в котором содержатся указатели на объекты класса `std::type_info` для всех целочисленных типов со знаком. Используя структуру `Length`, можно написать следующий код.

```
std::type_info* intsRtti[Length<SignedIntegrals>::value];
```

Во время вычислений на этапе компиляции в памяти будут размещены четыре элемента массива `intsRtti`[1].

3.5. Интермеццо

Впервые проблема шаблонных метапрограмм обсуждалась в книге Veldhuizen (1995). Затем эта тема глубоко изучалась в работе Czarnecki and Eisenecker (2000), которая содержала полную коллекцию имитаций выполнения операторов языка С++ на этапе компиляции программы.

Идея и реализация структуры `Length` напоминают классический пример рекурсии: алгоритм, вычисляющий длину односвязного списка структур. (Однако есть два существенных отличия: алгоритм для структуры `Length` выполняется на этапе компиляции и применяется к типам, а не к значениям.)

Возникает вопрос: можно ли разработать итеративный, а не рекурсивный вариант структуры `Length`? Помимо всего прочего, итерация более естественна для языка С++, чем рекурсия. Ответ на этот вопрос приведет нас к реализации других функциональных возможностей класса `Typelist`.

Ответ оказался отрицательным по весьма интересной причине.

Средства языка С++, предназначенные для программирования на этапе компиляции, состоят из шаблонов, целочисленных вычислений и определений типов (операторы `typedef`). Посмотрим, как работает каждый из этих инструментов.

Шаблоны — точнее, специализации шаблонов — представляют собой эквивалент операторов `if` на этапе компиляции. Как мы уже видели на примере реализации структуры `Length`, специализация шаблона позволяет отличать списки типов от других типов.

Целочисленные вычисления — это настоящие вычисления, которые осуществляются путем перехода от типов к значениям. Однако здесь есть одна особенность: все значения на этапе компиляции являются *неизменяемыми* (immutable). Определив целочисленную константу, скажем, перечислимое значение, программист не может его в дальнейшем изменять (например, присваивать одно значение другому).

Определения типов (операторы `typedef`) могут рассматриваться как введение констант, задающих имя типа. И вновь после определения все эти имена оказываются зафиксированными — в дальнейшем переопределить символ, введенный оператором `typedef`, невозможно.

Эти две особенности вычислений на этапе компиляции делают их принципиально несовместимыми с итерациями. Во время итераций вычисляется значение итератора,

---

[1] Этот массив можно инициализировать, не прибегая к повторению кода. Предлагаем читателю решить эту задачу самостоятельно.

причем оно может изменяться при выполнении некоторых условий. Поскольку на этапе компиляции переменных сущностей нет, выполнять итерации совершенно невозможно. Следовательно, хотя язык C++ в основном является императивным, любые вычисления на этапе компиляции должны опираться на методы, характерные для чисто функциональных языков, которые отличаются тем, что не могут изменять значения. Итак, без рекурсии не обойтись.

3.6. Индексированный доступ

К элементам списка типов желательно иметь индексированный доступ. Это позволит организовать линейный доступ к элементам, упрощая манипуляции со списками. Разумеется, как и все остальные сущности, с которыми мы работаем, индекс должен быть статической величиной.

Объявление шаблона для индексированной операции может выглядеть следующим образом.

```
template <class TList, unsigned int index> struct TypeAt;
```

Перейдем к определению алгоритма. Следует иметь в виду, что использовать изменяемые значения нельзя.

TypeAt

Входные данные: список типов TList, индекс i

Результат: внутренний тип Result

Если список TList не пуст и индекс i равен нулю,
  то класс Result — это голова списка TList.
Иначе,
  если список TList не пуст и индекс i не равен нулю,
    то класс Result получается путем применения алгоритма TypeAt
    к хвосту списка TList и индексу i-1.
  Иначе происходит выход за пределы допустимого диапазона изменения индекса,
  который порождает сообщение об ошибке на этапе компиляции.

Ниже приведено воплощение алгоритма TypeAt на языке C++.

```
template <class Head, class Tail>
struct TypeAt<Typelist<Head, Tail>, 0>
{
   typedef Head Result;
};
template <class Head, class Tail, unsigned int i>
struct TypeAt<Typelist<Head, Tail>, i>
{
   typedef typename TypeAt<Tail, i-1>::Result Result;
};
```

Если вы попробуете выйти за пределы допустимого диапазона изменения индекса, компилятор сообщит, что специализации TypeAt<NullType, *x*> не существует. Здесь символ *x* означает величину, на которую вы превысили размер списка. Это сообщение могло бы быть более информативным, но сойдет и так.

В библиотеке Loki (файл Typelist.h) определен вариант структуры TypeAt под названием TypeAtNonStrict. Эта структура реализует некоторые функциональные возможности структуры TypeAt с той лишь разницей, что выход за пределы допусти-

мого диапазона изменения индексов в ней запрещен менее строго и приводит не к порождению сообщения об ошибке на этапе компиляции, а к выдаче в качестве результата типа, заданного пользователем по умолчанию. Структура TypeAtNonStrict используется в обобщенной реализации обратного вызова, описанной в главе 5.

Время индексированного доступа к элементам списка типов прямо пропорционально размеру списка. Для списка значений этот метод неэффективен (по этой причине в классе std::list не определена функция operator[]). Однако для списков типов время тратится на этапе компиляции и его объем не имеет большого значения.[2]

3.7. Поиск элемента

Как найти нужный тип в списке типов? Попробуем реализовать алгоритм IndexOf, вычисляющий индекс типа в списке. Если тип не найден, результатом будет некоторое фиксированное число, например −1. Этот алгоритм представляет собой классическую рекурсивную реализацию линейного поиска.

IndexOF
Входные данные: список типов TList, тип T
Результат: внутренняя статическая константа value

Если список TList состоит из единственного типа NullType,
то значение value равно −1.
Иначе, если в голове списка TList находится тип T,
то значение value равно 0.
Иначе
вычислить результат алгоритма IndexOf, применяя его к хвосту списка Tlist и типу T, и присвоить его переменной temp.
Если значение temp равно -1, то значение value равно -1.
Иначе значение value равно 1 плюс temp.

Алгоритм IndexOf относительно прост. Особое внимание уделяется передаче значение value ("тип не найден") в качестве результата. Последняя ветвь алгоритма (вычисление значения value в зависимости от значения temp) представляет собой операцию над числами и выполняется с помощью условного оператора ?:. Ниже приводится реализация алгоритма.

```
template <class TList, class T> struct IndexOf;

template <class T>
struct IndexOF<NullType, T>
{
   enum { value = -1 };
};

template <class T, class Tail>
struct IndexOf<Typelist<T, Tail>, T>
```

[2] На самом деле при разработке больших проектов это не совсем так. Возможна ситуация, по крайней мере теоретически, когда сложные манипуляции со списками типов смогут существенно замедлить компиляцию программы. В любом случае программа, содержащая очень длинный список типов, либо очень долго выполняется (и тогда лучше смириться с долгой компиляцией), либо является слишком сложной (и тогда ее следует разработать заново).

```
{
   enum{ value = 0 };
};

template <class Head, class Tail, class T>
struct IndexOf<Typelist<Head, Tail>, T>
{
private:
   enum { temp = IndexOf<Tail, T>::value };
public:
   enum { value = temp == -1 ? -1 : 1 + temp };
};
```

3.8. Добавление элемента

Нам нужно иметь возможность добавлять в список типов новый тип или список типов. Поскольку модификация списка типов невозможна, будем выполнять "возврат по значению", создавая новый список типов, содержащий результат.

Append
Входные данные: список типов TList, тип или список типов T
Результат: определение внутреннего типа Result

Если список TList состоит из единственного типа NullType
и параметр T является типом NullType,
то класс Result является типом NullType.
Иначе, если список TList состоит из единственного типа NullType
и параметр T является отдельным типом (а не списком типов),
то класс Result является списком, состоящим из единственного элемента T.
Иначе, если список TList состоит из единственного типа NullType
и параметр T является списком типов,
то класс Result является типом T.
Иначе, если список TList не состоит из единственного типа NullType,
то класс Result является списком типов, головой которого является тип TList::Head, а хвостом — результат добавления списка T к элементу TList::Tail в качестве хвоста.

Этот алгоритм естественно выражается следующим кодом.

```
template <class TList, class T> struct Append;

template <> struct Append<NullType, NullType>
{
   typedef NullType Result;
};

template <class T> struct Append<NullType, T>
{
  typedef TYPELIST_1(T) Result;
};

template <class Head, class Tail>
struct Append<NullType, Typelist<Head, Tail> >
{
   typedef Typelist<Head, Tail> Result;
```

```
};

template <class Head, class Tail, class T>
struct Append<Typelist<Head, Tail>, T>
{
   typedef Typelist<Head,
         typename Append<Tail, T>::Result>
      Result;
};
```

Обратите внимание на то, как последняя частично специализированная версия структуры Append рекурсивно конкретизирует шаблон Append, передавая ему хвост списка и добавляемый тип.

Теперь для типов и списков типов определена унифицированная операция Append. Например, приведенный ниже оператор определяет список, состоящий из всех числовых типов со знаком, предусмотренных в языке C++.

```
typedef Append<SignedIntegrals,
     TYPELIST_3(float, double, long double)>::Result
  SignedTypes;
```

3.9. Удаление элемента

Рассмотрим теперь обратную операцию — удаление элемента из списка типов. У нас есть две возможности: удалить только первое вхождение или удалять все вхождения данного типа в списке.

Изучим только первый вариант.

Erase

Входные данные: список типов TList, тип T

Результат: определение внутреннего типа Result

Если список TList состоит из единственного типа NullType,
то класс Result является типом NullType.
Иначе, если тип T совпадает с типом TList::Head,
то класс Result является типом TList::Tail.
Иначе
класс Result является списком типов, голова которого является
типом TList::Head, а хвостом — результат применения алгоритма Erase
к хвосту TList:Tail с параметром T.

Вот как выглядит этот алгоритм на языке C++.

```
template <class TList, class T> struct Erase;
template <class T>                         // Специализация 1
struct Erase<NullType, T>
{
   typedef NullType Result;
};

template <class T, class Tail>             // Специализация 2
struct Erase<Typelist<T, Tail>, T>
{
   typedef Tail Result;
};

template <class Head, class Tail, class T > // Специализация 3
```

```
struct Erase<Typelist<Head, Tail>, T>
{
   typedef Typelist<Head,
         typename Erase<Tail, T>::Result>
      Result;
};
```

Как и для класса TypeAt, для этого шаблонного класса не предусмотрен вариант по умолчанию. Это означает, что класс Erase можно конкретизировать только определенными типами. Например, выражение Erase<double, int> вызовет ошибку компиляции, поскольку для него не существует ни одного соответствия. Кроме того, необходимо, чтобы первый параметр шаблонного класса Erase был списком типов.

Используя определение класса SingedTypes, можно записать следующее.

```
// Класс SomeSignedTypes содержит эквивалент класса
// TYPELIST_6(signed char, short int, int, long int,
// double, long double)
typedef Erase<SignedTypes, float>::Result SomeSignedTypes;
```

Рассмотрим рекурсивный алгоритм удаления элементов. Шаблон EraseAll удаляет все вхождения типа в список типов. Его реализация аналогична классу Erase, за одним исключением. При обнаружении типа, подлежащего удалению, алгоритм не останавливается, а продолжает искать и удалять соответствующие элементы из хвоста списка, рекурсивно применяя самого себя.

```
template <class TList, class T> struct EraseAll;

template <class T>
struct EraseAll<NullType, T>
{
   typedef NullType Result;
};

template <class T, clas Tail>
struct EraseAll<Typelist<T, Tail>, T>
{
   // Проходит вниз по списку, удаляя заданный тип
   typedef typename EraseAll<Tail, T>::Result Result;
};

template <class Head, class Tail, class T>
struct EraseAll<Typelist<Head, Tail>, T>
{
   // Проходит по списку, удаляя заданный тип
   typedef Typelist<Head
         typename EraseAll<Tail, T>::Result>
      Result;
};
```

3.10. Удаление дубликатов

Важной операцией над списками типов является удаление дубликатов. Ее цель — трансформировать список типов так, чтобы каждый тип в списке встречался лишь один раз. Например, из списка

```
TYPELIST_6(Widget, Button, Widget, TextField, ScrollBar, Button)
```

нужно получить список

```
TYPELIST_6(Widget, Button, TextField, ScrollBar)
```

Эта процедура немного сложнее, однако, как легко догадаться, нам поможет класс `Erase`.

`NoDuplicates`
Входные данные: список типов `TList`
Результат: определение внутреннего типа `Result`

Если список `TList` состоит из единственного типа `NullType`,
то класс `Result` является типом `NullType`.
Иначе
применить алгоритм `NoDuplicates` к списку `TList::Tail`,
получив временный список `L1`;
применить алгоритм `Erase` к списку `TList::Head`,
получив в результате список `L2`.
Класс `Result` — это список типов, голова которого является
списком `TList::Head`, а хвост — списком `L2`.

Вот как этот алгоритм переводится на язык C++.

```
template <class TList> struct NoDuplicates;

template <> struct NoDuplicates<NullType>
{
   typedef NullType Result;
};

template <class Head, class Tail>
struct NoDuplicates< Typelist<Head, Tail> >
{
private:
   typedef typename NoDuplicates<Tail>::Result L1;
   typedef typename Erase<L1, Head>::Result L2;
public:
   typedef Typelist<Head, L2> Result;
};
```

Почему мы воспользовались алгоритмом `Erase`, имея в своем распоряжении алгоритм `EraseAll`? Ведь мы хотим удалить все дубликаты данного типа, не правда ли? Ответ заключается в том, что алгоритм `Erase` применяется *после* рекурсивного вызова алгоритма `NoDuplicates`. Это означает, что мы удаляем из списка тип, у которого уже нет дубликата, поэтому в списке останется по крайней мере один экземпляр данного типа. Это довольно интересный пример рекурсивного программирования.

3.11. Замена элемента

Иногда вместо удаления нужно заменить какой-то элемент в списке типов. Как мы увидим в разделе 3.12, замена одного типа другим представляет собой важную строительную конструкцию для многих сложных идиом.

Допустим, что нам нужно заменить тип `T` типом `U` в списке типов `TList`.

`Replace`
Входные данные: список типов `TList`, тип `T` (подлежащий замене) и тип `U` (замена)
Результат: определение внутреннего типа `Result`

Если список TList состоит из единственного типа NullType,
то класс Result является типом NullType.
Иначе, если голова списка TList является типом T,
то класс Result представляет собой список типов,
у которого голова является типом U, а хвостом служит список
TList::Tail.
Иначе класс Result представляет собой список типов,
у которого голова является списком TList::Head,
а хвостом служит результат применения алгоритма Replace
к списку Tlist и параметрам T и U.

Вот как выглядит код этого рекурсивного алгоритма.

```
template <class TList, class T, class U> struct Replace;

template <class T, class U>
struct Replace<NullType, T, U>
{
   typedef NullType Result;
};

template <class T, class Tail, class U>
struct Replace<Typelist<T, Tail>, T, U>
{
   typedef Typelist<U, Tail> Result;
};

template <class Head, class Tail, class T, class U>
struct Replace<Typelist<Head, Tail>, T, U>
{
   typedef Typelist<Head,
         typename Replace<Tail, T, U>::Result
      Result;
};
```

Алгоритм ReplaceAll легко получить, изменив вторую специализацию так, чтобы в ней алгоритм рекурсивно применялся к списку Tail.

3.12. Частично упорядоченные списки типов

Допустим, нам нужно упорядочить список типов по отношению наследования. Например, мы бы хотели, чтобы производные типы предшествовали базовым. Пусть мы имеем иерархию классов, изображенную на рис. 3.1.

Предположим, что у нас есть список типов

```
TYPELIST_4(Widget, ScrollBar, Button, GraphicButton)
```

Наша задача — преобразовать его следующим образом:

```
TYPELIST_4(ScrollBar, GraphicButton, Button, Widget)
```

Следовательно, нам нужно переставить производные классы вперед, оставив порядок следования классов одинакового уровня (sibling) прежним.

На первый взгляд эта задача кажется надуманной, однако она имеет большое практическое значение. Просмотр упорядоченного списка типов, в котором производные типы стоят впереди, соответствует обходу иерархии классов снизу вверх. Ме-

ханизм двойной диспетчеризации, описанный в главе 11, применяет это важное свойство для распознавания информации о типах.

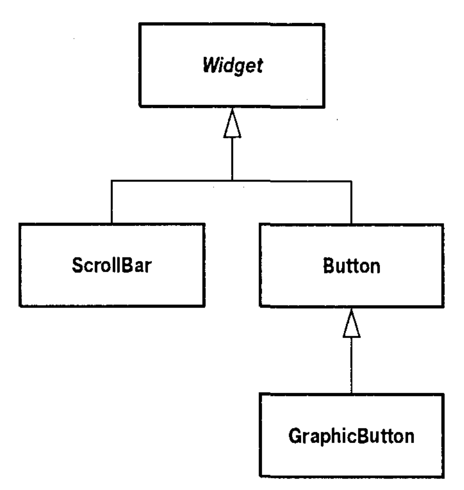


Рис. 3.1. Простая иерархия классов

Для упорядочения набора необходима соответствующая функция. У нас уже есть статический механизм распознавания наследования, детально описанный в главе 2 Напомним, что в нашем распоряжении есть удобный макрос `SUPERSUBCLASS(T, U)`, возвращающий значение `true`, если класс `U` является производным от класса `T`. Мы просто объединим механизм распознавания наследования со списками типов.

Применение к спискам типов полноценного алгоритма сортировки невозможно, поскольку у нас нет отношения полной упорядоченности между типами. По этой причине для классов невозможно создать эквивалент оператора сравнения `operator<`. Следовательно, нам придется использовать особенный алгоритм, переставляющий производные классы вперед и не изменяющий отношения между другими классами.

`DerivedToFront`

Входные данные: список типов `TList`

Результат: определение внутреннего типа `Result`

Если список `TList` состоит из единственного типа `NullType`
    то класс `Result` является типом `NullType`.
Иначе
    найти в списке `TList::Tail` класс самого нижнего уровня,
        производный от класса `TList::Head`. Сохранить его во
        временной переменной `TheMostDerived`.
    Заменить класс `TheMostDerived` в списке `TList::Tail`
        списком `TList::Head`, получив в результате список `L`.
    Сконструировать результат в виде списка типов, головой которого является
        класс `TheMostDerived`, а хвостом — список `L`.

Во время применения этого алгоритма к списку типов производные типы перемещаются на вершину, а базовые типы заталкиваются на дно.

В этом описании нет одной детали — алгоритма обнаружения производного класса самого нижнего уровня в заданном списке типов. Поскольку макрос `SUPERSUBCLASS`

во время компиляции порождает булевское значение, для решения этой задачи оказывается полезным небольшой шаблонный класс `Select` (также описанный в главе 2). Напомним, что этот шаблонный класс выбирает один из двух типов в зависимости от значения булевской константы.

Алгоритм `TheMostDerived` получает на вход список типов и тип `Base`, а возвращает — наиболее глубоко вложенный класс, производный от типа `Base`, находяшийся в списке (или, возможно, сам класс `Base`, если производных типов в списке нет).

`MostDerived`
Входные данные: список типов `TList`, тип `T`
Результат: определение внутреннего типа `Result`

Если список `TList` состоит из единственного типа `NullType`
то класс `Result` является типом `T`.
Иначе
применить алгоритм `MostDerived` к списку `TList::Tail` и типу `Base`.
Получить тип `Candidate`.
Если тип `TList::Head` является производным от типа `Candidate`,
то результатом является тип `TLIst::Head`.
Иначе результатом является тип `Candidate`.

Реализация алгоритма `MostDerived` выглядит следующим образом.

```
template <class TList, class T> struct MostDerived;

template <class T>
struct MostDerived<NullType, T>
{
   typedef T Result;
};

template <class Head, class Tail, class T>
struct MostDerived<Typelist<Head, Tail>, T>
{
private:
    typedef typename MostDerived<Tail, T>::Result Candidate;
public:
    typedef typename Select<
       SUPERSUBCLASS(Candidate, Head),
          Head, Candidate>::Result Result;
};
```

Алгоритм `DerivedToFront` использует алгоритм `MostDerived` в качестве элементарного алгоритма. Вот его реализация.

```
template <class T> struct DerivedToFDront;

template<>
struct DertivedToFront<NullType>
{
   typedef NullType Result;
};

template <class Head, class Tail>
struct DerivedToFront< Typelist<Head, Tail> >
{
```

```
private:
    typedef typename MostDerived<Tail, Head>::Result
        TheMostDerived;
    typedef typename Replace<Tail,
        TheMostDerived, Head>::Result L;
public:
    typedef Typelist<TheMostDerived, L> Result;
};
```

Эта сложная манипуляция со списком типов обладает замечательной силой. Преобразование `DerivedToFront` эффективно автоматизирует обработку типов, которую иначе пришлось бы долго и тщательно реализовывать вручную. Это напоминает автоматическую поддержку параллельных иерархий классов, не так ли?

3.13. Генерация класса на основе списка типов

Если до сих пор списки классов казались интересными, интригующими или даже ужасными, то это только цветочки. В этом разделе дается определение основных конструкций для генерации кода с помощью списков типов. Это означает, что мы вообще больше не будем писать программы, а заставим компилятор делать это за нас. Такие конструкции основаны на одной из самых мощных особенностей языка C++, которой нет ни в одном другом языке, — *шаблонных шаблонных параметрах* (template template parameters).

До сих пор манипуляции со списками типов не порождали никакого кода. В результате обработки возникал новый список типов, типы или числовые статические константы (как в классе Length). Перейдем к изучению процесса генерации реального кода, т.е. к тем сущностям, которые оставляют свой след в скомпилированном коде.

Объекты списков типов и здесь не нужны. Они не имеют никакого значения во время выполнения программы и никаких функциональных возможностей. При работе со списками типов особо важна возможность генерировать код на основе списка типов. Прикладные программисты иногда должны наполнять класс каким-то кодом — сигнатурами виртуальных функций, объявлениями переменных или реализациями функций. Способ этого заполнения определяется списком типов. Попытаемся автоматизировать этот процесс.

Поскольку в языке C++ нет статических итераций или рекурсивных макросов, задача добавления кода для каждого типа, содержащегося в списке, довольно сложна. Можно применить частичную шаблонную специализацию в сочетании с описанными выше алгоритмами, но реализация этого решения будет слишком запутанной и сложной. Решить эту задачу нам поможет библиотека Loki.

3.13.1. Генерация распределенных иерархий

Мощные шаблоны библиотеки Loki облегчают задачу построения классов путем применения каждого типа, содержащегося в списке типов, к основному шаблону, предоставленному пользователем. Таким образом, запутанный процесс распределения типов, указанных в списке, по программе пользователя инкапсулирован в библиотеке. Пользователь должен лишь определить простой шаблон с одним параметром.

Этот шаблонный библиотечный класс называется `GetScatterHierarchy`. Несмотря на то что его определение достаточно просто, этот класс невероятно эффективен. Вот его определение.[3]

---

[3] Это одна из ситуаций, когда предпочтительнее сначала описать идею и лишь затем — ее потенциальные приложения (в отличие от обычного порядка решения задач).

```
template <class TList, template <class> class Unit>
class GenScatterHierarchy;
// Специализация класса GenScatterHierarchy:
// преобразование класа Typelist в класс Unit
template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<TypeList<T1, T2>, Unit>
   : public GenScatterHierarchy<T1, Unit>,
     public GenScatterHierarchy<T1, Unit>
{
public:
   typedef Typelist<T1, T2> TList;
   typedef GenScatterHierarchy<T1, Unit> LeftBase;
   typedef GenScatterHierarchy<T2, Unit> RightBase;
};

// Передача атомарного типа (не списка типов) классу Unit
template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtopmicType>
{
   typedef Unit<AtomicType> LeftBase;
};

// С классом NullType не делаем ничего
template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>
{
};
```

Шаблонный шаблонный параметр работает так, как и ожидалось (глава 1). Шаблонный класс Unit передается классу GenScatterHierarchy в качестве второго аргумента. Класс GenScatterHierarchy использует свой шаблонный шаблонный параметр Unit как обычный шаблонный класс с одним шаблонным параметром. Новые возможности появляются благодаря тому, что пользователь класса GenScatterHierarchy может передавать ему свой собственный шаблон.

Что делает класс GenScatterHierarchy? Если его первый аргумент является атомарным типом (в противоположность списку типов), класс GenScatterHierarchy передает его классу Unit и наследует свойства результирующего класса Unit<T>. Если первый аргумент является списком типов TList, класс GenScatterHierarchy сводится (recurses) к классам GenScatterHierarchy<TList::Head, Unit> и GenScatterHierarchy<TList::Tail, Unit> и наследует их свойства. Класс GenScatterHierarchy<NullType, Unit> является пустым.

В итоге процесс конкретизации класса GenScatterHierarchy завершается наследованием класса Unit, конкретизированного *каждым типом из списка типов*. В качестве примера рассмотрим следующий код.

```
template <class T>
struct Holder
{
   T value_;
};

typedef GetScatterHierarchy<
      TYPELIST_3(int, string, Widget),
      Holder>
    WidgetInfo;
```

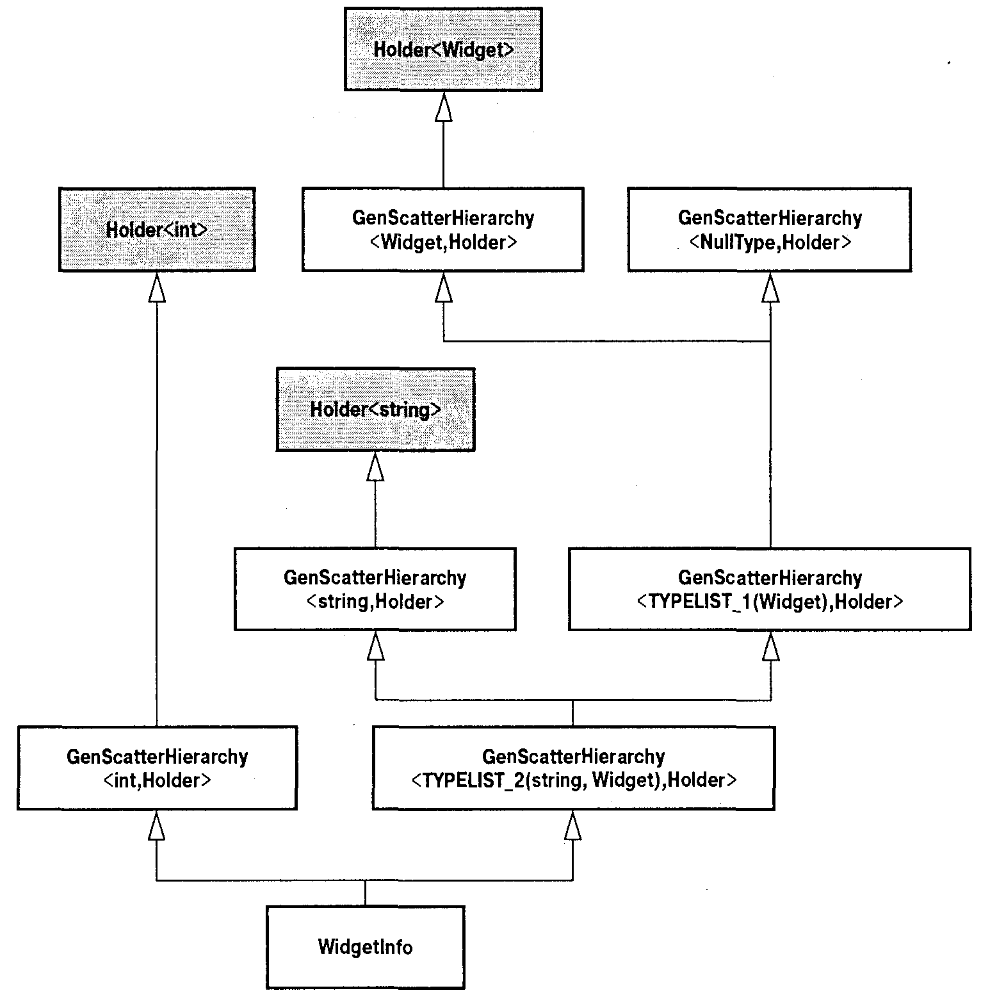


Рис. 3.2. Структура наследования класса WidgetInfo

Иерархия наследования, порожденная классом `WidgetInfo`, показана на рис. 3.2. Мы будем называть такие иерархии классов *распределенными* (scattered), поскольку типы в списке типов распределены по разным корневым классам. В этом заключается сущность класса `GenScatterHierarchy` — он генерирует иерархию классов для пользователя, многократно конкретизируя шаблонный класс, предоставленный ему в качестве модели. Затем он собирает все сгенерированные классы в один, в данном случае — класс `WidgetInfo`.

В результате наследования классы `Holder<int>`, `Holder<string>`, `Holder<Widget>` и `WidgetInfo` имеют одну переменную-член `value_type` для каждого типа, указанного в списке типов. На рис. 3.3 показана бинарная схема объекта класса `WidgetInfo`. Предполагается, что такие пустые классы, как `GenScatterHierarchy<NullType, Holder>`, игнорируются и не занимают места в составном объекте.

Объекты класса WidgetInfo позволяют делать интересные вещи. Например, можно обратиться к объекту класса string, хранящемуся в объекте класса WidgeInfo, написав следующий код.

```
WidgetInfo obj;
string name = (static_cast<Holder<string>&>(obj)).value_;
```

Явное приведение типов позволяет избежать неоднозначности имени value_. В противном случае компилятор не сможет определить, на какой член value_ вы ссылаетесь.

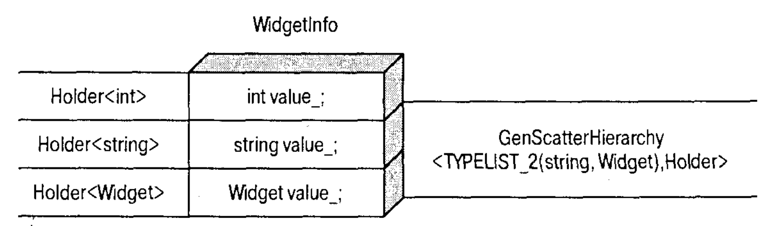


Рис. 3.3. Схема распределения памяти для объекта класса WidgetInfo

Это приведение типов выглядит ужасно, поэтому мы попытаемся облегчить работу с классом GenScatterHierarchy, снабдив его несколькими удобными функциями доступа. Например, было бы прекрасно иметь доступ к члену класса по его типу. Это довольно просто.

```
// Класс FieldTraits описан в файле HierarchyGenerators.h
template <class T, class H>
typename Private::FieldTraits<H>::Rebind<T>::Result&
Field(H& obj)
{
   return obj;
}
```

Работа функции Field основана на неявном преобразовании производного типа в базовый. При вызове Field<Widget>(obj) (где obj должен иметь тип WidgetInfo) компилятор распознает, что Holder<Widget> — это базовый класс по отношению к классу WidgetInfo, и просто вернет ссылку на эту часть составного объекта.

Почему функция Field является частью пространства имен, а не функцией-членом? Потому что такой высокий уровень обобщенного программирования вынуждает очень осторожно обращаться с именами. Представьте, например, что в классе Unit определен символ с именем Field. Если бы в классе GenScatterHierarchy была функция-член Field, она бы была замаскирована функцией Field, являющейся членом класса Unit. Это вызвало бы массу недоразумений.

У функции Field есть один недостаток: ее нельзя применять, если в списке типов есть дубликаты. Рассмотрим немного измененное определение класса WidgetInfo.

```
typedef GenScatterHierarchy<
   TYPELIST_4(int, int, string, Widget),
   Value>
WidgetInfo;
```

Теперь в классе WidgetInfo есть два члена value_, имеющих тип int. Если вызвать функцию Field<int> из объекта класса WidgetInfo, компилятор пожалуется на неоднозначность. Устранить эту неоднозначность нелегко, поскольку класс WidgetInfo дважды наследует свойства класса Holder<Int>, причем разными путями, как показано на рис. 3.4.

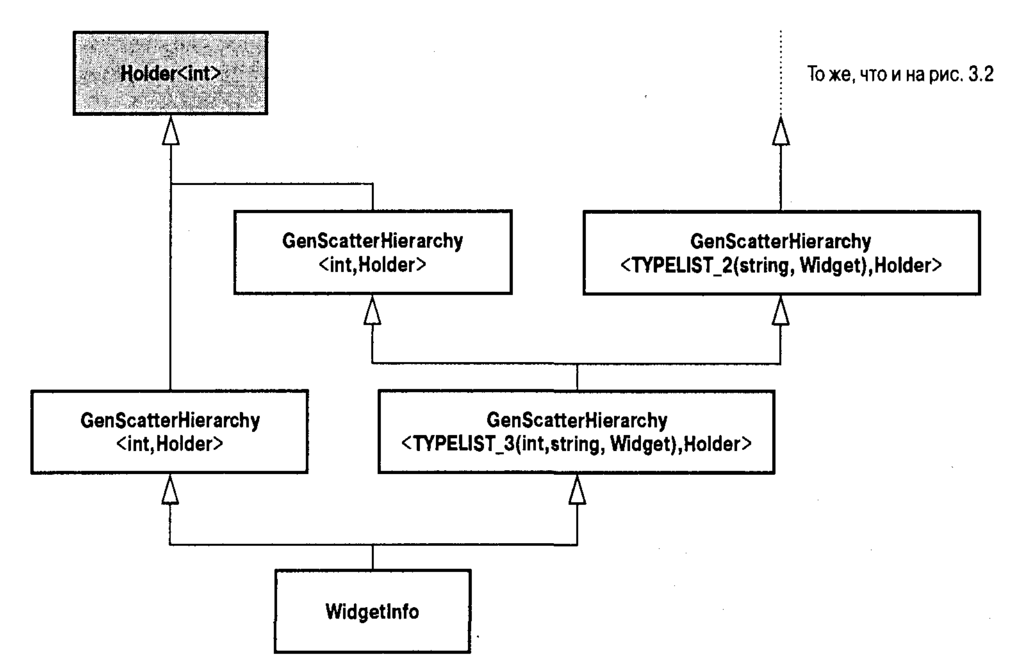


Рис. 3.4. Класс WidgetInfo наследует свойства класса Holder<int> дважды

Нам необходимо средство, позволяющее выбирать поля в конкретизации класса `GenScatterHierarchy` на основе *позиционного индекса* (positional index). Если бы мы могли ссылаться на каждое из этих двух полей, имеющих тип `int`, в списке типов (т.е. `Field<0>(obj)` и `Field<1>(obj)`), то неоднозначность была бы устранена.

Попробуем реализовать функцию доступа к полю по его позиции в списке типов. Нам нужно выполнить статическую диспетчеризацию для поля, имеющего индекс 0, что соответствует голове списка типов, и поля с ненулевым индексом, соответствующего хвосту списка. Это довольно легко сделать с помощью небольшого шаблонного класса `Int2Type`, определенного в главе 2. Напомним, что класс `Int2Type` просто преобразовывает каждую отдельную целочисленную константу в отдельный тип. Кроме того, как показано ниже, для передачи полученного результата используется класс `Type2Type`.

```
template <class H, typename R>
inline R& FieldHelper(H& obj, Type2Type<R>, Int2Type<0>)
{
   typename H::LeftBase& subobj = obj;
   return subobj;
}

template <class H, typename R, int i>
inline R& FieldHelper(H& obj, Type2Type<R> tt, Int2Type<i>)
{
   typename H::RightBase& subobj = obj;
   return FieldHelper(subobj, tt, Int2Type<i-1>());
}
// Класс FieldTraits описан в файле HierarchyGenerators.h
template <int i, class H>
template Private::FieldTraits<H>::At<i>::Result&
Field(H& obj)
```

```
{
   typedef typename Private::FieldTraits<H>::At<i>::Result
      Result;
   return FieldHelper(obj, Type2Type<Result>(), Int2Type<i>());
}
```

Остается понять, как написана такая реализация, но, к счастью, это достаточно легко объяснить. Всю работу выполняют две перегруженные функции FieldHelper. Первая из них получает параметр, имеющий тип Int2Type<0>, а вторая — параметр Itn2Type<*любое целое число*>. Следовательно, первая перегруженная функция возвращает значение, соответствующее классу Unit<T1>&, а вторая — тип, имеющий указанный индекс в списке типов. Для того чтобы определить, какой тип следует вернуть, функция Field использует вспомогательный шаблонный класс FieldTraits. Функция Field возвращает этот тип функции FieldHelper через класс Type2Type.

Вторая перегруженная функция FieldHelper рекурсивно вызывает саму себя, передавая правого предка класса GenScatterHierarchy и тип Int2Type<index-1>, поскольку поле N в списке типов является полем N−1 в хвосте этого списка, если N не равно нулю. (Действительно, случай, когда N равно нулю, обрабатывается первой перегруженной функцией.)

Чтобы упростить интерфейс, нам нужно предусмотреть в классе Field две дополнительные функции: константные версии двух определенных выше функций Field. Эти функции очень похожи на свои неконстантные прототипы, но, в отличие от них, принимают и возвращают ссылки на константные типы.

Функция Field намного облегчает использование класса GenScatterHierarchy. Теперь можно написать следующий код.

```
WidgetInfo obj;
...
int x = Field<0>(obj).value_; // Первый целый тип
int x = Field<1>(obj).value_; // Второй целый тип
```

Шаблонный класс GenScatterHierarchy очень удобен для генерации сложных классов на основе списков простых типов. Класс GenScatterHierarchy можно применять для генерации виртуальных функций для каждого типа в списке. В главе 9, посвященной абстрактным фабрикам, этот класс используется для генерации абстрактных производящих функций, работающих со списками типов. В этой главе также показано, как реализуются иерархии, порожденные классом GenScatterHierarchy.

3.13.2. Генерация кортежей

Иногда возникает необходимость создать небольшую структуру, состоящую из безымянных полей, известную в некоторых языках (например, в языке ML) под названием *кортеж* (tuple). Создание кортежей на языке C++ впервые было описано в работе Якко Ярви (Jakko Järvi, 1999a), а затем уточнено в работе (Järvi and Powell, 1999b).

Что такое кортеж? Рассмотрим следующий пример.

```
template <class T>
struct Holder
{
   T value_;
};

typedef GenScatterHierarchy<
   TYPELIST_3(int, int, int),
```

```
   Holder>
Point3D;
```

Работать с классом Point3D довольно трудно, поскольку после каждой функции доступа к полям необходимо указывать суффикс .value_. Нам нужно создать структуру, похожую на класс GenScatterHierarchy, в которой функции доступа Field возвращают ссылки непосредственно на члены value_. Это значит, что функция Field<n> должна возвращать не Holder<int>, а int&.

В библиотеке Loki определен класс Tuple, реализованный аналогично классу GenScatterHierarchy, но предоставляющий прямой доступ к полям. Этот класс работает следующим образом.

```
typedef Tuple<TYPELIST_3(int, int, int)>
   Point3D;
Point3D pt;
Field<0>(pt) = 0;
Field<1>(pt) = 100;
Field<2>(pt) = 300;
```

Кортежи полезны для создания небольших безымянных структур, не имеющих функций-членов. Например, с их помощью можно возвращать из функции сразу несколько значений.

```
Tuple<TYPELIST_3(int, int, int)>
GetWindowPlacement(Window&);
```

Фиктивная функция GetWindowPlacement позволяет пользователям узнавать координаты окна и его положение в стеке окон, используя один вызов функции. При этом разработчик библиотеки не обязан предусматривать отдельную структуру для кортежа, состоящего из трех целых чисел.

Другие функции, работающие с кортежами, можно найти в файле Tuple.h библиотеки Loki.

3.13.3. Генерация линейных иерархий

Рассмотрим следующий простой шаблонный класс, определяющий интерфейс обработчика события. В нем определяется только функция-член OnEvent.

```
template <class T>
class EventHandler
{
public:
   virtual void OnEvent(const T&, int eventId) = 0;
   virtual void ~EventHandler() {}
};
```

В классе EventHandler определен виртуальный деструктор, не представляющий для нас интереса, но тем не менее необходимый (причины будут указаны в главе 4).

Шаблонный класс GenScatterHierarchy можно использовать для того, чтобы распространить класс EventHandler на каждый тип в списке.

```
typedef GenScatterHierarchy
<
   TYPELIST_3(Window, Button, ScrollBar),
   EventHandler
>
WidgetEventHandler;
```

У класса GenScatterHierarchy есть недостаток — он использует множественное наследование. Если размер кода очень важен, этот класс может стать непригодным, поскольку класс WidgetEventHandler содержит три указателя на виртуальные таблицы[4], по одной для каждой конкретизации класса EventHandler. Если величина sizeof(EventHandler) равна 4 байт, то величина sizeof(WidgetEventHandler) будет равна уже 12 байт, возрастая по мере добавления в список новых типов. Наиболее эффективно было бы объявлять все виртуальные функции внутри класса WidgetEventHandler, правда, это не позволило бы генерировать код.

Линейная иерархия наследования позволяет разложить класс WidgetEventHandler на классы, по одному на каждую виртуальную функцию, как показано на рис. 3.5. При использовании простого наследования класс WidgetEventHandler может иметь только один указатель на виртуальную таблицу, что максимально повышает его эффективность с точки зрения размера.

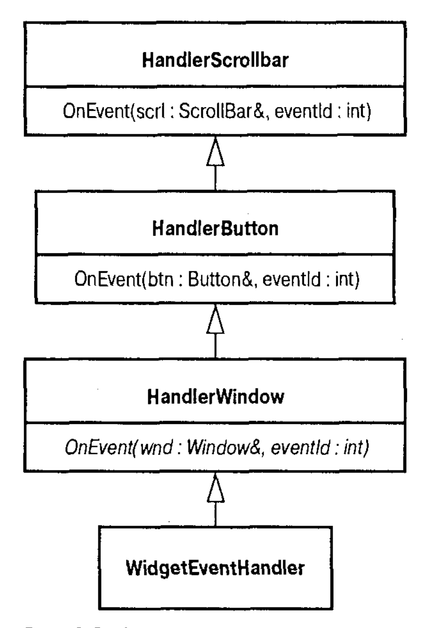


Рис. 3.5. Оптимальная по размеру структура класса WidgetEventHandler

Как создать механизм, автоматически генерирующий такую структуру? Для этого предназначен рекурсивный механизм, аналогичный классу GenScatterHierarchy. Однако есть одно отличие. Шаблонный класс, определенный пользователем, теперь должен принимать *два* шаблонных параметра. Один из них является текущим типом в списке типов, как и у класса GenScatterHierarchy, а второй — это базовый класс, от которого происходит конкретизация. Второй шаблонный параметр необходим, поскольку, как показано на рис. 3.5, код, определенный пользователем, теперь включается *в середину* иерархии классов, а не только в его корни (как это было в классе GenScatterHierarchy).

[4] Реализация не обязана использовать виртуальные таблицы, но большинство программистов все же применяют их. Описание виртуальных таблиц дано в работе Lippman (1994).

Напишем рекурсивный шаблонный класс GenLinearHierarchy. Он похож на класс GenScatterHierarchy, но отношение наследования и шаблонный код, определенный пользователем, в нем обрабатываются иначе.

```
template
<
   class TList,
   template <class AtomicType,class Base> class Unit,
   class Root = EmptyType // Класс EmptyType описан в главе 2
>
class GenLinearHierarchy;

template
<
   class T1,
   class T2,
   template <class, class> class Unit,
   class Root
>
class GenLinearHierarchy<Typelist<T1, T2>, Unit, Root>
   : public Unit< T1, GenLinearHierarchy<T2, Unit, Root> >
{
};

template
<
   class T,
   template <class, class> class Unit,
   class Root
>
class GenLinearHierarchy<TYPELIST_1(T), Unit, Root>
   : public Unit<T, Root>
{
};
```

Этот код немного сложнее, чем у класса GenScatterHierarchy, но структура иерархии класса GenLinearHierarchy намного проще. Следуя пословице “лучше один раз увидеть, чем сто раз услышать” (“image is worth 1,024 words”), посмотрим на рис. 3.6, на котором изображена иерархия классов, порожденная следующим кодом.

```
template <class T, class Base>
class EventHandler : public Base
{
public:
   virtual void OnEvent(T& obj, int eventId);
};

typedef GenLinearHierarchy
<
   TYPELIST_3(Window, Button, ScrollBar),
   EventHandler
>
MyEventHandler;
```

В сочетании с классом EventHandler класс GewtLinearHierarchy определяет линейную иерархию простого наследования. Каждый узел в этой иерархии определяет одну чисто виртуальную функцию, как это предусмотрено для класса EventHandler. Следовательно, в классе EventHandler определены три виртуальные функции, как и требовалось. Класс GenLinearHierarchy выдвигает новое требование к своему шаблонному

параметру: класс Unit (в нашем примере — класс EventHandler) должен получать второй шаблонный параметр и наследовать его свойства. В качестве компенсации класс GenLinearHierarchy выполняет сложную работу по генерации иерархии классов.

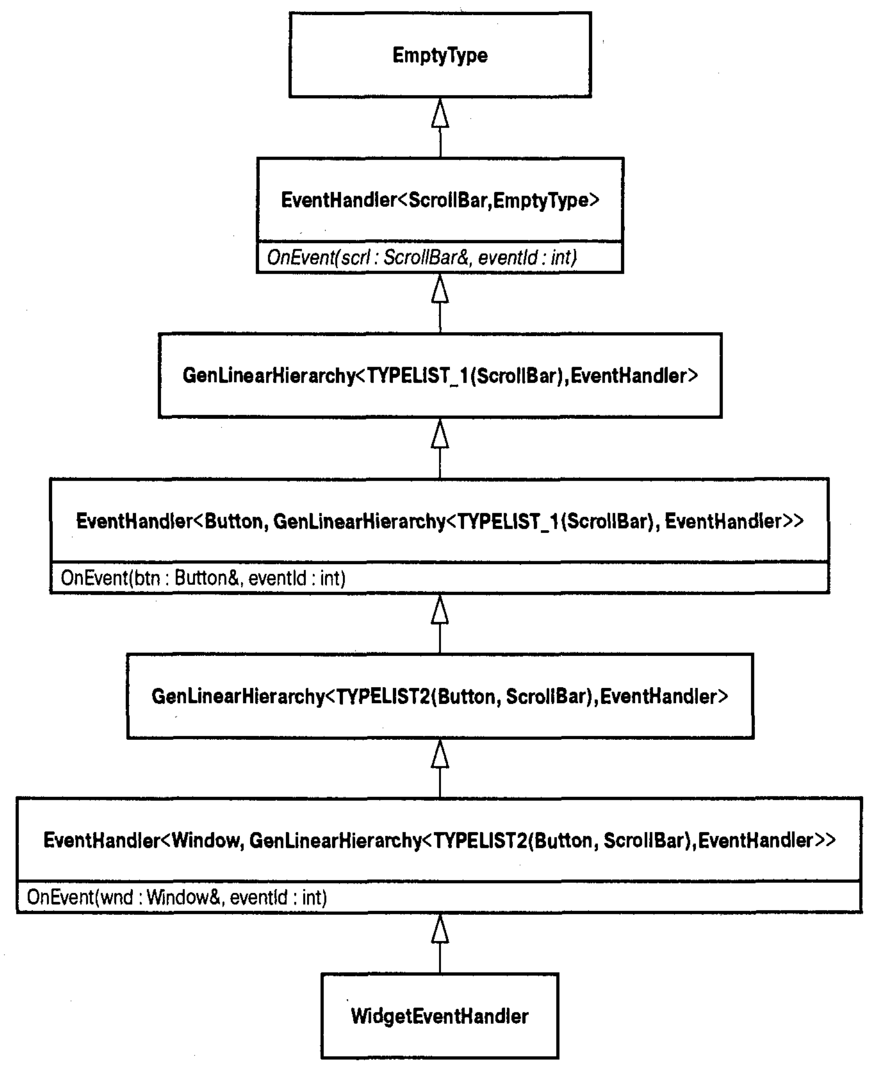


Рис. 3.6. Иерархия классов, порожденная шаблонным классом GenLinearHierarchy

Классы GenScatterHierarchy и GenLinearHierarchy прекрасно работают в тандеме. В большинстве случаев интерфейс можно генерировать с помощью класса GenScatterHierarchy, а реализацию — с помощью класса GenLinearHierarchy. Конкретные примеры использования этих двух генераторов приводятся в главах 9 и 10.

3.14. Резюме

Списки типов представляют собой важный метод обобщенного программирования. Они предоставляют создателям библиотек новые возможности: выражать и манипули-

ровать неограниченно большими наборами типов, генерировать структуры данных и код на основе этих коллекций и т.д.

На этапе компиляции к спискам типов можно применять большинство элементарных функций, предусмотренных для обычных списков: добавлять, удалять и заменять элементы, удалять дубликаты, обращаться к элементам, осуществлять их поиск и даже выполнять их частичное упорядочение. Код, реализующий манипуляции со списками типов, ограничен чисто функциональным стилем, поскольку на этапе компиляции не существует изменяемых величин — типы и статические константы, будучи однажды определенными, уже никогда не изменяются. По этой причине большинство операций над списками типов основано на рекурсивных шаблонах и сопоставлении образцов с помощью частичной шаблонной специализации.

Списки типов полезны, когда приходится писать один и тот же код — декларативный или императивный — для каждого отдельного типа. Они позволяют абстрагировать и обобщать сущности, которые невозможно абстрагировать и обобщать с помощью других приемов программирования. По этой причине списки типов представляют собой средство для создания совершенно новых идиом и реализаций библиотек, как мы увидим в главах 9 и 10.

Библиотека Loki содержит два мощных примитива для автоматической генерации иерархий классов: шаблонные классы `GenScatterHierarchy` и `GenLinearHierarchy`. Они генерируют две основные структуры классов: распределенную (см. рис. 3.2) и линейную (см. рис. 3.6). Линейная иерархия классов наиболее эффективна с точки зрения размера. Распределенная иерархия классов обладает полезным свойством: все конкретизации шаблона, определенного пользователем (передаваемого в качестве аргумента классу `GenScatterHierarchy`), представляют собой корни финального класса, как показано на рис. 3.2.

3.15. Краткое описание класса Typelist

- *Заголовочный файл*: `Typelist.h`.
- Все сущности класса находятся в пространстве имен `Loki::TL`.
- Определен шаблонный класс `Typelist<Head, Tail>`.
- Создание списка типов: определены макросы от `TYPELIST_1` до `TYPELIST_50`. Количество принимаемых ими параметров указано в их имени.
- Пользователь может повысить верхний предел макросов (50).

```
#define TYPELIST_51(T1, повторять вплоть до T51) \
    TYPELIST(T1, TYPELIST(T2, повторять вплоть до T51)>
```

- По соглашению первым элементом списков типов всегда является простой тип (не список типов), а хвост может быть либо списком типов, либо классом `NullType`.
- В заголовочном файле определен набор примитивов, оперирующих со списками типов. По соглашению все эти примитивы возвращают результат в виде определения вложенного (внутреннего) открытого типа под названием `Result`. Если результатом работы примитива является значение, оно имеет имя `value`.
- Примитивы описаны в табл. 3.1.
- Краткое описание шаблонного класса `GenScatterHierarchy`:

```
template <class TList, template <class> class Unit>
class GenScatterHierarchy;
```

- Шаблонный класс GenScatterHierarchy порождает иерархию, конкретизирующую класс Unit для каждого типа, указанного в списке типов TList. Конкретизация класса GenScatterHierarchy прямо или косвенно наследует свойства класса Unit<T> для каждого типа T из списка типов TList.
- Структура иерархии, порожденной классом GenScatterHierarchy, изображена на рис. 3.2.
- Краткое описание шаблонного класса GenLinearHierarchy:

```
template <class TList, template <class, class> class Unit>
class GenLinearHierarchy;
```

- Шаблонный класс GenLinearHierarchy порождает линейную иерархию, изображенную на рис. 3.6.
- Шаблонный класс GenLinearHierarchy конкретизирует класс Unit, передавая каждый тип из списка типов TList в качестве первого шаблонного параметра этого класса. ***Обратите внимание***: класс Unit должен создаваться на основе второго шаблонного параметра с помощью открытого наследования.
- Перегруженные функции Field обеспечивают доступ к узлам иерархии по типу и по индексу.
- Функция Field<Type>(obj) возвращает ссылку на конкретизацию класса Unit, соответствующую указанному типу Type.
- Функция Field<index>(obj) возвращает ссылку на конкретизацию класса Unit, соответствующую типу, позиция которого в списке типов задается целочисленной константой index.

Таблица 3.1. Статические алгоритмы работы со списками типов

Имя примитива	Описание примитива
Length<TList>	Вычисляет длину списка типов Tlist
TypeAt<TList, idx>	Возвращает тип, занимающий заданную позицию в списке типов TList (считая от нуля). Если индекс больше длины списка или равен ей, возникает ошибка компиляции
TypeAtNonStrict<TList, idx>	Возвращает тип, занимающий заданную позицию в списке типов TList (считая от нуля). Если индекс больше длины списка или равен ей, возвращается тп NullType
IndexOf<TList, T>	Возвращает индекс первого вхождения типа T в список TList. Если тип не найден, возвращает число −1
Append<TList, T>	Добавляет в список TList новый тип или список типов
Erase<TList, T>	Удаляет первое вхождение в список TList типа T (если оно есть)
EraseAll<TList, T>	Удаляет все вхождения в список TList типа T (если они есть)

Окончание табл. 3.1

Имя примитива	Описание примитива
NoDuplicates<TList, T>	Удаляет из списка Tlist все дубликаты
Replace<TList, T>	Заменяет первое вхождение в список TList типа T (если оно есть) типом U
ReplaceAll<TList, T>	Заменяет все вхождения в список TList типа T (если они есть) типом U
MostDerived<TList, T>	Определяет наиболее глубоко вложенный тип, производный от типа T. Если такого типа нет, возвращается тип T
DerivedToFront<TList, T>	Передвигает наиболее глубоко вложенные производные типы в голову списка типов Tlist

4

Размещение в памяти небольших объектов

В этой главе обсуждаются вопросы разработки и реализации механизма быстрого распределения памяти (allocator) для небольших объектов. Использование таких распределителей позволяет компенсировать дополнительные затраты ресурсов, происходящие при динамическом размещении объектов.

В разных местах библиотеки Loki используются очень маленькие объекты — их размер может не превышать нескольких байтов. В главе 5, посвященной обобщенным функторам, и главе 7, описывающей интеллектуальные указатели, маленькие объекты используются очень широко. По разным причинам, среди которых наиболее важной является их полиморфное поведение, эти маленькие объекты нельзя хранить в стеке.

Для работы с динамической памятью в языке C++ есть два основных инструмента — операторы `new` и `delete`. Проблема заключается в том, что эти операторы универсальны и неэффективно работают при динамическом размещении маленьких объектов. Чтобы проиллюстрировать, насколько плохо они работают с маленькими объектами, заметим, что некоторые стандартные распределители динамической памяти иногда работают на порядок медленнее и занимают в два раза больше памяти, чем распределители, описанные в этой главе.

"Источник всех бед — ранняя оптимизация", — заявил Дональд Кнут (Donald Knith). С другой стороны, Лен Латтанци (Len Lattanzi) утверждает, что "поздняя пессимизация не приводит ни к чему хорошему". Пессимизация только на один порядок во время выполнения программы таких основных объектов, как функторы, интеллектуальные указатели или строки, легко может угробить весь проект. ("Пессимизация" — генерирование программы, которая заведомо хуже ее оптимизированного варианта. — *Прим. ред.*) Выгоды, которые приносит дешевое и быстрое динамическое распределение памяти для маленьких объектов, могут быть огромными, поскольку она позволяет применять совершенные технологии, не опасаясь значительных потерь производительности. Эти рассуждения являются достаточно сильным побудительным мотивом для разработки способов оптимального размещения маленьких объектов в динамической памяти.

Авторы многих книг, посвященных языку C++, например, Саттер (Sutter, 2000) и Мейерс (Meyers, 1998a), упоминают о полезности разработки своих собственных специализированных средств распределения памяти. Мейерс, описав какую-либо реализацию, оставляет некоторые детали "в виде упражнения для самостоятельной работы", а Саттер отсылает читателей к "учебникам по C++ или программированию вообще". Издание, которое вы держите в руках, не претендует на то, чтобы стать вашей настольной книгой. Однако в

этой главе мы опустимся на уровень “железа” и реализуем свой собственный распределитель памяти в соответствии со стандартом языка C++ в мельчайших деталях.

В этой главе описаны нюансы, связанные с настройкой механизмов распределения памяти. В ней рассмотрен сверхмощный механизм распределения памяти для маленьких объектов из библиотеки Loki, рабочая лошадка для интеллектуальных указателей и обобщенных функторов.

4.1. Стандартный механизм распределения динамической памяти

По неизвестным причинам стандартный механизм распределения динамической памяти в языке C++ работает крайне медленно. Возможно, это связано с тем, что обычно он реализуется как тонкая оболочка вокруг распределителя динамической памяти языка С (`malloc/realloc/free`), который неэффективно работает с небольшими участками памяти. В программах, написанных на языке С, обычно не применяются идиомы, позволяющие многократно выделять и освобождать небольшие участки памяти. Вместо этого программы на языке С размещают в памяти средние и большие объекты (сотни и тысячи байтов), поэтому работа функций `malloc/free` оптимизирована именно для них.

Кроме этого, универсальность механизма распределения памяти в языке C++ стала причиной крайней неэффективности использования памяти, выделяемой для маленьких объектов. Стандартный распределитель управляет кучей, что часто вынуждает его занимать дополнительную память. Обычно вспомогательная память занимает от 4 до 32 байт для каждого блока, выделенного с помощью оператора new. При выделении блоков памяти размером 1024 байт дополнительные затраты незначительны (от 0,4 % до 3 %). Однако, если возникает необходимость разместить в памяти объекты размером 8 байт, дополнительные затраты памяти составят от 50 % до 400 %. Эти числа достаточно велики, чтобы заставить программиста задуматься о том, как ему разместить в памяти большое количество маленьких объектов.

В языке C++ динамическое распределение памяти имеет большое значение. Динамический полиморфизм часто ассоциируется именно с динамическим распределением памяти. Такие стратегии, как “идиома Pimple” (the pimple idiom) (Sutter, 2000), изначально ориентируются на распределение динамической, а не статической памяти. (Название идиомы представляет собой аббревиатуру выражения “a **P**ointer to the **IMPLE**mentation class” — “указатель на класс реализации”. — *Прим. ред.*)

Следовательно, низкая производительность стандартного механизма распределения памяти делает его камнем преткновения на пути создания эффективных программ на языке C++. Бывалые программисты на C++ инстинктивно избегают конструкций, ориентированных на использование динамической памяти, поскольку они по опыту знают о связанных с этим дополнительных затратах. Таким образом, у программистов возникает некий психологический барьер.

4.2. Как работает стандартный механизм распределения динамической памяти

Изучать тенденции, связанные с использованием памяти, как показал Кнут (Knuth, 1998), очень интересно. Он разработал много основных стратегий распределения памяти, а после опубликования его работы их появилось еще больше.

Как работает стандартный механизм распределения памяти? Он управляет пулом байтов и может выделять из него участки памяти любого размера. В качестве вспомогательной структуры может применяться простой блок управления.

```
struct MemControlBlock
{
   std::size_t size_;
   bool available_;
};
```

Память, которой управляет структура MemControlBlock, располагается сразу за ним, а ее объем, выраженный в байтах, определяется переменной size_. Вслед за этим участком памяти располагается следующая структура MemControlBlock и т.д.

Во время запуска программы в начале пула располагается только одна структура MemControlBlock, которая управляет всей памятью как одним большим участком. Это — корневой блок управления, который никогда никуда не перемещается. На рис. 4.1 показана схема распределения памяти для пула, имеющего объем 1 Мбайт, в момент запуска программы.

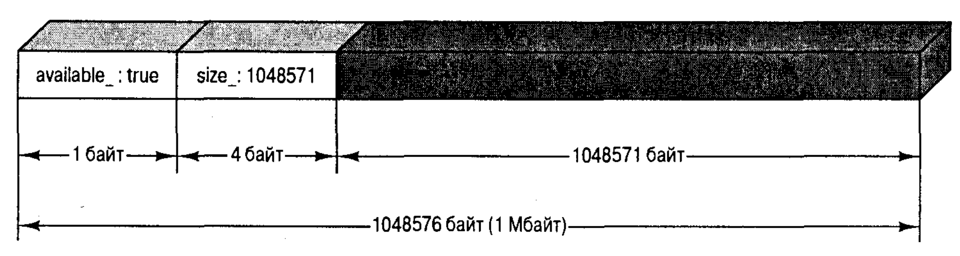


Рис. 4.1. Карта распределения памяти в момент запуска программы

При каждом запросе на выделение памяти выполняется линейный поиск подходящего блока, размер которого либо равен требуемому, либо превышает его. Поразительно, как много существует стратегий, позволяющих удовлетворить эти запросы, и как странно все они работают! Среди подходящих блоков памяти можно найти первый, наилучший, наихудший и даже случайно выбранный. Как ни странно, наихудший блок оказывается лучше наилучшего — как вам этот оксюморон?!

Каждое удаление объекта из памяти влечет за собой новый линейный поиск блока, предшествующего удаляемому, и последующее выравнивание его размера.

Как видим, эта стратегия не слишком эффективна с точки зрения затрат времени, однако дополнительные затраты памяти довольно малы — в каждом блоке нужно разместить две переменные типа size_t и bool соответственно. Иногда можно даже сэкономить один бит на каждой переменной типа size_t, использовав его для размещения переменной available_. Сжатая структура MemControlBlock имеет следующий вид.

```
// Код, зависящий от платформы и компилятора
struct MemControlBlock
{
   std::size_t size_ : 31;
   bool available_ : 1;
};
```

Если в каждом объекте MemControlBlock хранить указатели на предыдущий и следующий объекты этого типа, время, необходимое для освобождения памяти, может стать постоянным. Это происходит потому, что удаляемый блок связан со смежными

блоками и может их выравнивать соответствующим образом. Для этого необходима такая структура блока управления.

```
struct MemControlBlock
{
   bool available_;
   MemControlBlock* prev;
   MemControlBlock* next_;
}
```

На рис. 4.2 показано распределение пула памяти в виде дважды связанного списка блоков. Переменная-член size_ больше не нужна, поскольку размер блока можно вычислить с помощью выражения this->next_ - this. Дополнительные затраты памяти, связанные с необходимостью хранить два указателя и переменную типа bool, по-прежнему существуют. (Как и ранее, можно прибегнуть к системно-зависимым трюкам и запаковать булевскую переменную в один из указателей.)

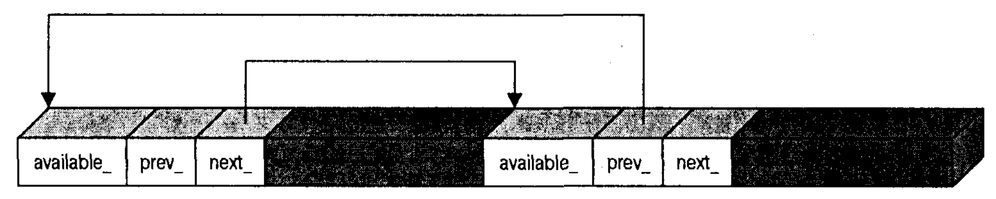


Рис. 4.2. Распределитель памяти с постоянным временем удаления

Время, затрачиваемое этим распределителем памяти, по-прежнему прямо пропорционально количеству блоков. Для того чтобы уменьшить эти затраты, существует много искусных трюков, каждый из которых основан на собственной системе компромиссов. Интересно, что совершенной стратегии распределения памяти до сих пор не существует; каждая из них использует дополнительную память, что снижает ее эффективность.

Мы не ставим перед собой задачу усовершенствовать стандартный механизм распределения памяти, а сосредоточимся на специализированных распределителях, которые лучше всего работают с небольшими объектами.

4.3. Распределитель памяти для небольших объектов

Распределитель памяти для небольших объектов работает с четырехслойной структурой, показанной на рис. 4.3. Ее верхние слои используют функциональные возможности, предоставляемые нижними слоями. На дне структуры расположен тип Chunk. Каждый объект типа Chunk содержит участок памяти (chunk), состоящий из целого числа блоков фиксированного размера, и управляет им. Объекты типа Chunk позволяют выделять и освобождать блоки памяти. Если в объекте типа Chunk не осталось свободных блоков, функция распределения памяти вернет нуль.

В следующем слое расположен класс FixedAllocator. Объекты этого класса используют участки памяти в качестве строительных блоков. Основное предназначение класса FixedAllocator — удовлетворять запросы на выделение памяти, объем которой превышает размер участка. Для этого объекты класса FixedAllocator образовывают массив объектов типа Chunk. Если поступил запрос на выделение памяти и все существующие участки памяти заняты, объект класса FixedAllocator создает новый объект типа Chunk и записывает его в массив. Затем он выполняет запрос на выделение памяти, переадресовывая его этому объекту.

SmallObject	* Предоставляет функциональные возможности на уровне объектов * Прозрачен — классы наследуют свойства только объектов класса SmallObject
SmallObjAllocator	* Позволяет размещать в памяти небольшие объекты разного размера * Конфигурацию параметров можно изменять
FixedAllocator	* Размещает объекты только одного фиксированного размера
Chunk	* Размещает объекты только одного фиксированного размера * Максимально возможное количество размещаемых объектов фиксировано

Рис. 4.3. Многослойная структура распределителя памяти для небольших объектов

Класс SmallObjAllocator выполняет общие функции, связанные с выделением и освобождением памяти. Объекты этого класса состоят из нескольких объектов класса FixedAllocator, каждый из которых специализирован для выделения объектов определенного размера. В зависимости от требуемого количества байтов объекты класса SmallObjAllocator направляют запрос одному из объектов класса FixedAllocator или стандартному оператору ::operator new, если запрашиваемый объем памяти слишком велик.

На верхнем уровне расположен класс SmallObject, представляющий собой оболочку класса FixedAllocator. Он инкапсулирует все функциональные возможности, связанные с распределением динамической памяти, и предоставляет их другим классам. Класс SmallObject перегружает операторы new и delete и передает их объектам класса SmallObjAllocator. Таким образом, объекты, создаваемые программистом, могут наследовать функции для работы с динамической памятью от класса SmallObject.

Кроме того, можно непосредственно использовать классы SmallObjAllocator и FixedAllocator. (Класс Chunk слишком примитивен и ненадежен, поэтому он определяется в закрытом разделе класса FixedAllocator.) Большей частью, однако, программы пользователей просто наследуют свойства базового класса SmallObjAllocator, приобретая функциональные возможности для распределения динамической памяти. Интерфейс этого класса достаточно прост.

4.4. Класс Chunk

Каждый объект класса Chunk содержит участок памяти, состоящий из фиксированного числа блоков, и управляет им. Размер и количество блоков указываются при создании объекта класса Chunk. Объекты типа Chunk позволяют выделять и освобождать блоки памяти. Если в объекте типа Chunk не осталось свободных блоков, функция распределения памяти вернет нуль.

Ниже приведено определение класса Chunk.

```
// В классе нет закрытых полей — класс Chunk представляет собой
// структуру старых простых данных (Plain Old Data - POD).
// Структура определяется внутри класса FixedAllocator
// и управляется только им.
struct Chunk
{
   void Init(std::size_t blockSize, unsigned char blocks);
   void* Allocate(std::size_t blockSize);
   void Deallocate(void* p, std::size_t blockSize);
   unsigned char* pData_;
```

```
    unsigned char
        firstAvailableBlock_,
        blocksAvailable_;
};
```

Кроме указателя на управляемый участок памяти объект класса Chunk хранит следующие целочисленные величины.

- Переменная firstAvailableBlock_ хранит индекс первого доступного блока внутри порции памяти.
- Переменная blocksAvailable_ представляет собой количество блоков, доступных в данной порции памяти.

Интерфейс класса Chunk очень прост. Функция Init инициализирует объект класса Chunk, а функция Release — освобождает занятую память. Функция Allocate выделяет блок памяти, а функция Deallocate освобождает его. Функциям Allocate и Deallocate нужно передавать размер памяти, поскольку в самом объекте класса Chunk эта величина не хранится, так как на верхних уровнях структуры размер блока неизвестен. Если бы в классе Chunk была предусмотрена переменная-член blockSize_, это было бы пустой тратой времени и памяти. Не забывайте, что мы находимся на самом дне структуры — здесь все имеет значение. По соображениям эффективности в классе Chunk не определены ни конструкторы, ни деструктор, ни оператор присваивания. Попытка определить семантику копирования на этом уровне снизила бы эффективность работы верхних уровней структуры, на которых объекты класса Chunk хранятся внутри векторов.

Структура Chunk демонстрирует важные ограничения. Поскольку переменные blocksAvailable_ и firstAvailableBlock_ имеют тип unsigned char, порция не может содержать больше 255 блоков (на компьютерах, где символы состоят из 8 бит). Как мы вскоре убедимся, это решение вполне удачно и позволяет избежать многих неприятностей.

Перейдем теперь к более интересной теме. Блок может быть занят или свободен. Все, что необходимо сохранить, записывается в свободные блоки. В первом байте свободного блока хранится индекс следующего свободного блока. Поскольку первый доступный индекс хранится в переменной firstAvailableBlock_, мы получаем полноценный односвязный список свободных блоков, не используя дополнительной памяти.

В момент инициализации объект класса Chunk выглядит, как показано на рис. 4.4. Код его инициализации приведен ниже.

```
void Chunk::Init(std::size_t blockSize, unsigned char blocks)
{
    pData_=new unsigned char[blockSize * blocks];
    firstAvailableBlock_ = 0;
    blocksAvailable_ = blocks;
    unsigned char i = 0;
    unsigned char* p = pData_;
    for (; i != blocks; p += blockSize)
    {
        *p = ++i;
    }
}
```

Этот односвязный список, встроенный в структуру данных, представляет собой большое достижение. Он позволяет быстро и эффективно находить свободный блок внутри порции без дополнительных затрат памяти, причем на размещение и удаление блока тратится постоянное время, что исключительно важно.

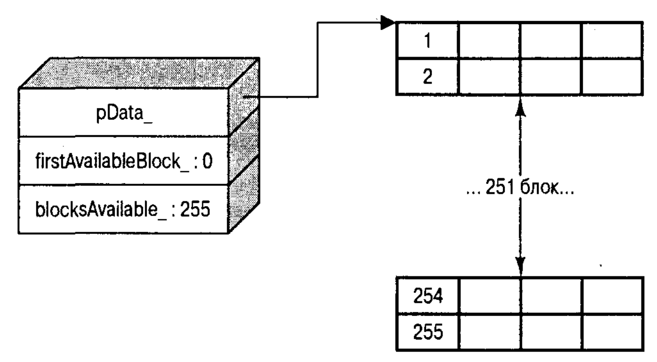


Рис. 4.4. Объект класса Chunk, состоящий из 255 блоков по 4 блока каждый

Посмотрим теперь, почему количество блоков ограничено величиной, соответствующей максимальному значению переменной типа unsigned char. Допустим, что мы используем переменную более широкого типа, например unsigned short, которая на многих компьютерах занимает 2 байт. В этом случае перед нами возникают две проблемы — большая и маленькая.

- Мы не можем размещать блоки, размер которых меньше величины sizeof(unsigned short), что очень неудобно, поскольку мы разрабатываем механизм распределения памяти для *небольших* объектов. Это маленькая проблема.
- Мы сталкиваемся с вопросами выравнивания участков памяти. Вообразите, что мы создали механизм распределения для 5-байтовых блоков. В этом случае приведение указателя, ссылающегося на 5-байтовый блок, к типу unsigned int приведет к непредсказуемым последствиям. Это большая проблема.

Решить эти проблемы довольно просто — нужно использовать тип unsigned char в качестве типа для "тайного индекса" ("stealth index"). Символьный тип по определению имеет размер, равный 1 байт, поэтому никаких проблем, связанных с выравниванием участков памяти не возникает, поскольку даже указатель на отдельную ячейку памяти ссылается на переменную типа unsigned char.

Это приводит к ограничению максимального количества блоков в объекте класса Chunk, поскольку в нем не может быть больше, чем UNCHAR_MAX блоков (в большинстве систем эта величина равна 255). Это вполне приемлемо, даже если размер блока действительно мал, например, от 1 до 4 байт. Это относится и к более крупным блокам, поскольку в любом случае объекты класса Chunk по определению должны быть маленькими.

Функция распределения памяти извлекает блок, на который ссылается переменная firstAvailableBlock_, и устанавливает ее на следующий свободный блок — типичная операция над списками.

```
void* Chunk::Allocate(std::size_t blockSize)
{
   if (!blocksAvailable_) return 0;
   unsigned char* pResult =
      pData_ + (firstAvailableBlock_ * blockSize);
   // Присвоение переменной firstAvailableBlock_
   // указателя на следующий свободный блок
   firstAvailableBlock_ = *pResult;
   --blocksAvailable_;
```

```
    return pResult;
}
```

Итак, функция `Chunk::Allocate` состоит из одной операции сравнения, одной операции доступа по индексу, двух операций вычитания, присваивания и оператора декрементации — вполне приемлемо. Еще более важно то, что в этой функции не выполняется операция поиска. На рис. 4.5 показано состояние объекта класса `Chunk` после выполнения первой операции распределения памяти.

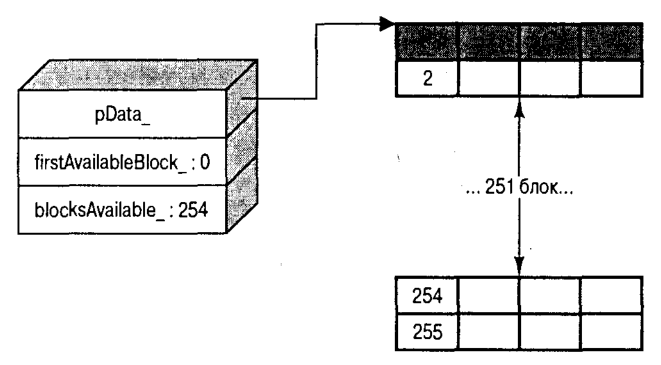


Рис. 4.5. Состояние объекта класса Chunk после выполнения первой операции распределения памяти. Занятая память выделена серым цветом

Функция освобождения памяти работает в точности наоборот. Она возвращает блок в список свободных блоков и увеличивает значение переменной `blocksAvailable_`. Не забудьте о необходимости передавать размер блока в качестве параметра функции `Deallocate`, поскольку объекту класса `Chunk` он неизвестен.

```
void Chunk::Deallocate(void* p, std::size_t blockSize)
{
    assert(p >= pData_);
    unsigned char* toRelease = static_cast<unsigned char*>(p);
    // Проверка выравнивания
    assert((toRelease - pData_) % blockSize == 0);
    *toRelease = firstAvailableBlock_;
    firstAvailableBlock_ = static_cast<unsigned char>(
        (toRelease - pData_) / blockSize);
    // Проверка усечения
    assert(firstAvailableBlock_ ==
        (toRelease - pData_) / blockSize);
    ++blocksAvailable_;
}
```

Функция освобождения памяти небогата содержанием, однако в ней содержится довольно много диагностических высказываний (которые по-прежнему не перехватывают все ошибочные ситуации). Структура `Chunk` вполне соответствует традиционным механизмам распределения памяти, принятым в языках C и C++. Если вы передадите функции `Chunk::Deallocate` неверный указатель, приготовьтесь к худшему.

4.5. Класс FixedAllocator

На следующем уровне механизма распределения памяти для небольших объектов находится класс `FixedAllocator`. Этот класс предназначен для распределения памяти

при работе с блоками фиксированного размера, причем этот размер не обязан соответствовать размеру порции.

Для этого объект класса FixedAllocator образует вектор объектов типа Chunk. При поступлении запроса на распределение памяти объект класса FixedAllocator отыскивает подходящий объект класса Chunk. Если все порции заняты, объект класса FixedAllocator создает новый объект класса Chunk. Вот как выглядит эта часть определения класса.

```
class FixedAllocator
{
   ...
private:
   std::size_t blockSize_;
   unsigned char numBlocks_;
   typedef std::vector<Chunk> Chunks;
   Chunks chunks_;
   Chunk* allocChunk_;
   Chunk* deallocChunk_;
};
```

Чтобы ускорить работу, класс FixedAllocator не перебирает элементы вектора chunks_ в поисках подходящего участка памяти при каждом запросе. Вместо этого в объекте хранится указатель на последний объект класса Chunk, который использовался при распределении памяти (переменная allocChunk_). При поступлении нового запроса функция FixedAllocate сначала проверяет, достаточно ли памяти в порции, на которую ссылается указатель allocChunk_. Если этот объект удовлетворяет требованиям запроса, то используется именно он. Если нет, выполняется линейный поиск (и, возможно, в вектор chunks_ записывается новый объект класса Chunk). В любом случае указатель allocChunk_ устанавливается на найденный или вновь созданный объект класса Chunk. Такой подход повышает вероятность того, что в следующий раз распределение памяти будет выполнено быстро. Вот как выглядит код этого алгоритма.

```
void* FixedAllocator::Allocate()
{
   if (allocChunk_ == 0 ||
     allocChunk_->blocksAvailable_ == 0)
   {
     // В данном объекте памяти недостаточно.
     // Попробуйте найти другой.
     Chunks::iterator i = chunks_.begin();
     for (;; +=i)
     {
        if (i == chunks_.end())
        {
           // Все объекты заполнены — добавляем новый.
           chunks_.reserve(chunks_.size()+1);
           Chunk newChunk;
           newChunk.Init(blockSize_, numBlocks_);
           chunks_.push_back(newChunk);
           allocChunk_ = &chunks_.back();
           deallocChunk_ = &chunks_.back();
           break;
        }
        if (i->blocksAvailable_ > 0)
        {
           // Объект найден
           allocChunk_ = &*i;
```

```
            break;
        }
    }
}
assert(allocChunk_ != 0);
assert(allocChunk_->blocksAvailable_ > 0);
return allocChunk_->Allocate(blockSize_);
}
```

Придерживаясь этой стратегии, класс FixedAllocator удовлетворяет большинство запросов за одно и то же время. При этом могут возникать некоторые задержки, обусловленные поиском и добавлением новых блоков. Иногда эта схема может работать неэффективно. Однако на практике такие случаи встречаются редко. Не забывайте, что у каждого механизма распределения памяти есть своя ахиллесова пята.

Освобождение памяти — более сложная задача, поскольку определенная часть информации отсутствует. У нас есть лишь указатель на освобождаемую область памяти, и нам неизвестно, какому именно объекту типа Chunk он принадлежит. Мы можем просмотреть содержимое вектора chunks_, проверяя, лежит ли данный указатель в диапазоне от pData_ до pData_ + blockSize_ * numBlocks_. Если да, то указатель следует передать функции-члену Deallocate соответствующего объекта класса Chunk. Очевидно, что этот поиск требует затрат времени. Несмотря на то что операция выделения памяти выполняется быстро, процедура удаления занимает линейное время. Итак, нужно как-то ускорить этот процесс.

Для этого можно применить вспомогательный *кэш* (cache memory). Когда пользователь освобождает блок, используя вызов FixedAllocator::Deallocate(p), объект класса FixedAllocator не передает указатель p обратно соответствующему объекту класса Chunk, а записывает указатель p во внутреннюю память — кэш, в котором хранятся свободные блоки. При поступлении нового запроса объект класса FixedAllocator сначала просматривает кэш. Если он не пуст, объект извлекает из кэша последний доступный указатель и немедленно возвращает его обратно. Это очень быстрая операция. Только когда кэш опустошен, объект класса FixedAllocator вынужден выполнить стандартную процедуру распределения памяти для объекта класса Chunk. Эта многообещающая стратегия в некоторых достаточно распространенных случаях работает плохо.

При распределении памяти для небольших объектов возможны такие ситуации.

- *Распределение памяти для большого массива данных.* В памяти одновременно размещается большое количество мелких объектов. Это происходит, например, при инициализации набора указателей на маленькие объекты.
- *Удаление в том же порядке.* Большое количество мелких объектов удаляется в том же порядке, в котором они были размещены в памяти. Это происходит при разрушении большинства контейнеров из стандартной библиотеки шаблонов STL[1].
- *Удаление в обратном порядке.* Большое количество небольших объектов удаляется в порядке, обратном порядку их размещения в памяти. Для программ, напи-

---

[1] Стандарт языка C++ не определяет порядок уничтожения объектов, содержащихся в стандартном контейнере. Следовательно, каждый программист, занимающийся реализацией языка, может выбирать этот порядок по своему усмотрению. Обычно контейнеры разрушаются с помощью простого прямого повторения. Однако в некоторых реализациях разработчики сочли "более естественным" уничтожать объекты в обратном порядке. В обоснование этого приводится факт, что в языке C++ объекты разрушаются именно в обратном порядке.

санных на языке C++, это вполне естественная ситуация при вызове функций, работающих с небольшими объектами. Аргументы функции и временные переменные стека соответствуют именно этому случаю.

- *Произвольное размещение и удаление.* Объекты создаются и удаляются без определенного порядка. Это происходит, когда при выполнении программы время от времени возникает необходимость в небольших объектах.

Стратегия кэширования очень хорошо соответствует ситуации, при каждой объекты размещаются и удаляются в произвольном порядке. Однако при размещении и удалении большого массива данных кэширование оказывается бесполезным. Хуже того, оно замедляет процесс удаления объектов, поскольку каждый раз тратит дополнительное время на очистку буфера.[2]

Лучше всего применять стратегию удаления объектов, совпадающую со стратегией их размещения в памяти. Переменная-член `FixedAllocator::deallocChunk_` указывает на последний объект класса `Chunk`, который был использован для удаления из памяти. Как только поступает запрос на удаление, эта переменная проверяется первой. Затем, если она ссылается на неверный объект типа `Chunk`, функция `Deallocate` начинает линейный поиск, обнаруживает требуемый объект и обновляет переменную `deallocChunk_`.

Есть два важных приема, позволяющих ускорить работу функции `Deallocate` в указанных выше ситуациях. Во-первых, функция `Deallocate` выполняет поиск подходящего объекта класса `Chunk`, начиная с окрестности указателя `deallocChunk_`. Это значит, что вектор `chunks_` просматривается, начиная с переменной `deallocChunk_`, а следующими проверяются итераторы, находящиеся выше и ниже. Это значительно повышает скорость удаления большого массива данных при любом порядке удаления (прямом или обратном). Во время размещения большого массива данных функция `Allocate` добавляет объекты класса `Chunk` по порядку. Во время удаления либо указатель `deallocChunk_` сразу попадает в точку, либо правильный объект будет обнаружен на следующем шаге.

Во втором трюке следует избегать крайних вариантов. Допустим, оба указателя `allocChunk_` и `deallocChunk_` ссылаются на последний объект в векторе, и свободного места в этом множестве не осталось. Представьте, что произойдет, если будет выполняться следующий код.

```
for (...)
{
   // Некоторые интеллектуальные указатели используют свой
   // механизм распределения памяти для небольших объектов
   // (глава 7)
   SmartPtr p;
      ... используется указатель p ...
}
```

При каждом проходе цикла создается и уничтожается объект класса `SmartPtr`. При создании объекта, поскольку памяти больше нет, функция `FixedAllocator::Allocate` создает новый объект класса `Chunk` и заносит его в вектор `chunks_`. При разрушении объекта функция `FixedAllocator::Deallocate` обнаруживает пус-

---

[2] Мне не удалось создать разумную схему кэширования, которая одинаково хорошо работала бы при удалении объектов в прямом и обратном порядке. Кэширование наносит вред либо одному, либо другому процессу. Поскольку обе ситуации часто встречаются в реальных программах, кэширование лучше не применять.

той блок и освобождает его. Эти затратные процедуры повторяются на каждой итерации цикла for.

Такая неэффективная работа недопустима. Следовательно, во время удаления объект класса Chunk должен освобождаться, только если есть *два* пустых объекта этого класса. При наличии только одного пустого участка он быстро перемещается в конец вектора chunks_. Таким образом мы избегаем выполнения затратных операций vector<Chunk>::erase, всегда удаляя лишь последний элемент.

Разумеется, могут возникнуть ситуации, при которых такой подход неприменим. При размещении в цикле вектора объектов класса SmartPtr определенного размера может возникнуть та же проблема. Однако такие ситуации встречаются все реже и реже. Кроме того, как указывалось во введении, *любой* механизм распределения памяти в определенных ситуациях может оказаться хуже остальных.

Выбранная стратегия удаления объектов хорошо соответствует произвольному порядку их создания. Даже если данные размещались без определенного порядка, программы стремятся достичь определенной *локальности* (locality), каждый раз обращаясь к небольшой порции данных. Указатели allocChunk_ и deallocChunk_ в этих ситуациях работают безукоризненно, поскольку при последующем размещении и удалении объектов эти указатели действуют как кэш.

Итак, у нас есть класс FixedAllocator, достаточно эффективно удовлетворяющий запросы на распределение памяти фиксированного размера как с точки зрения затрачиваемого времени, так и с точки зрения использования памяти. Класс FixedAllocator оптимизирован для работы с небольшими объектами.

4.6. Класс SmallObjAllocator

Третий уровень нашей архитектуры механизма распределения памяти состоит из класса SmallObjAllocator, позволяющего размещать в памяти объекты любого размера, объединяя в одно целое несколько объектов класса FixedAllocator. Получив запрос на распределение памяти, объект класса SmallObjAllocator либо переадресует его наиболее подходящему объекту класса FixedAllocator, либо передаст его оператору ::operator new.

Ниже приведено краткое описание класса SmallObjAllocator. Объяснения даются после кода.

```
class SmallObjAllocator
{
public:
   SmallObjAllocator(
      std::size_t chunkSize,
      std::size_t maxObjectSize);
   void* Allocate(std::size_t numBytes);
   void Deallocate(void* p, std::size_t size);
   ...
private:
   std::vector<FixedAllocator> pool_;
   ...
};
```

Конструктор получает два параметра, определяющих конфигурацию объекта класса SmallObjAllocator. Параметр size задает размер участка памяти, принятый по умолчанию (длина каждого объекта класса Chunk, выраженная в байтах), а параметр maxObjectSize задает максимально возможный размер объектов, которые еще могут

считаться "небольшими". Объект класса SmallObjAllocator переадресовывает запросы на выделение блоков, размер которых превышает величину maxObjectSize, непосредственно оператору ::operator new.

Довольно необычно, что функция Deallocate получает размер объекта, подлежащего удалению, в качестве аргумента. Дело в том, что это позволяет ускорить процесс освобождения памяти. В противном случае функция SmallObjAllocator::Deallocate должна была бы искать среди всех объектов класса FixedAllocator тот, которому принадлежит данный указатель. Это слишком затратная процедура, поэтому объекту класса SmallObjAllocator нужно передавать размер удаляемого блока. Как мы увидим в следующей главе, эта задача изящно решается самим компилятором.

Какое отображение существует между размером блока в объекте FixedAllocator и вектором pool_? Иными словами, как по заданному размеру отыскать соответствующий объект класса FixedAllocator, который выполняет распределение памяти и ее освобождение?

Простая и эффективная идея заключается в том, чтобы элемент pool_[i] обрабатывал объекты размера i. Вектор pool_ инициализируется с размером maxObjectSize, а затем инициализируются все объекты класса FixedAllocator, содержащиеся в нем. Если поступает запрос на выделение памяти размером numBytes, объект класса SmallObjAllocator передает его либо элементу pool_[numBytes] (эта операция имеет постоянное время выполнения), либо оператору ::operator new.

Однако это решение не настолько удачное, как кажется на первый взгляд. В конкретных ситуациях нам могут понадобиться распределители памяти только одного определенного размера, например 4 байт, и никакие другие объекты не нужны. В этом случае придется распределять память для 64 и большего количества элементов вектора pool_, хотя на самом деле используются только два элемента.

Выравнивание размеров и заполнение пустот еще больше снижают эффективное использование памяти. Например, многие компиляторы дополняют все типы, определенные пользователем, до размера, кратного заданному числу (2, 4 и т.д.). Например, если компилятор дополняет все структуры до размера, кратного 4, то используется только 25% элементов вектора pool_, остальные представляют собой балласт.

Намного лучше пожертвовать скоростью просмотра, но сэкономить память[3]. Мы будем хранить объекты класса FixedAllocator только для тех размеров памяти, которые требуются хотя бы один раз. Таким образом, вектор pool_ может содержать много объектов разного размера, не слишком увеличиваясь. Для того чтобы ускорить просмотр, вектор pool_ хранится отсортированным по размерам блоков.

Кроме того, ускорить просмотр можно с помощью стратегии, которая уже применялась в классе FixedAllocator. Объект класса SmallObjAllocator хранит указатель на последний объект класса FixedAllocator, использованный при освобождении памяти. Ниже приведен полный список переменных-членов класса SmallAllocator.

```
class SmallObjAllocator
{
   ...
private:
   std::vector<FixedAllocator> pool_;
   FixedAllocator* pLastAlloc_;
```

[3] Фактически в современных системах экономия памяти ведет к ускорению работы. Это происходит благодаря огромной разнице между скоростью работы основной памяти (большой и медленной) и кэш-памяти (маленькой и быстрой).

```
    FixedAllocator* pLastDealloc_;
};
```

При поступлении запроса на выделение памяти сначала проверяется указатель pLastAlloc_. Если его размер не соответствует ожидаемому, функция SmallObjAllocator::Allocate выполняет бинарный поиск в массиве pool_. Освобождение памяти осуществляется примерно так же. Единственное отличие заключается в том, что функция SmallObjAllocator::Allocate может закончить свою работу, вставив в массив pool_ новый объект класса FixedAllocator.

Как уже отмечалось при обсуждении класса FixedAllocator, эта простая схема кэширования предназначена для многократного создания и удаления объектов за постоянное время.

4.7. Трюк

На последнем уровне нашей архитектуры расположен класс SmallObject, базовый класс, инкапсулирующий функциональные возможности, предоставленные классом SmallObjAllocator.

Класс SmallObject перегружает операторы new и delete. Таким образом, при создании объекта класса, производного от класса SmallObject, в действие вступает перегрузка, направляющая запрос объекту класса FixedAllocator.

Определение класса SmallObject довольно простое, но интересное.

```
class SmallObject
{
public:
    static void* operator new(std::size_t size);
    static void operator delete(void* p, std::size_t size);
    virtual ~SmallObject() {}
};
```

Класс SmallObject выглядит вполне нормально, за исключением одной маленькой детали. Во многих книгах, посвященных языку C++, например, в учебнике Sutter (2000), говорится, что при перегрузке оператора delete его единственным аргументом должен быть указатель типа void.

В языке C++ есть одна лазейка, которая нас очень интересует. (Напомним, что мы разработали класс SmallObjAllocator так, чтобы размер освобождаемого блока передавался как аргумент.) В стандартном варианте оператор delete можно перегрузить двумя способами. Первый из них выглядит так.

```
void operator delete(void* p);
```

Второй способ таков.

```
void operator delete(void* p, std::size_t size);
```

Эта тема очень подробно изложена в книге Sutter (2000).

Если используется первый способ, размер удаляемого блока памяти игнорируется. Однако этот размер нам очень нужен, поскольку его следует передать объекту класса SmallObjAllocator. Следовательно, нам подходит лишь второй способ перегрузки.

Как заставить компилятор автоматически определять размер блока? На первый взгляд кажется, что для этого понадобится дополнительная память для каждого объекта, хотя именно этого мы и стремились избежать.

Однако на самом деле никакой дополнительной памяти не требуется. Рассмотрим следующий код.

```
class Base
{
   int a_[100];
public:
   virtual ~Base() {}
};
class Derived : public Base
{
   int b_[200];
public:
   virtual ~Derived() {}
};
...
Base* p = new Derived;
delete p;
```

Объекты классов Base и Derived имеют разные размеры. Для того чтобы избежать лишних затрат памяти, связанных с необходимостью хранить размер фактического объекта, на который ссылается указатель p, компилятор прибегает к следующему трюку: он генерирует код, распознающий размер на лету. Этого можно достичь с помощью следующих четырех приемов. (Вообразите на несколько минут, что мы перевоплотились в разработчиков компилятора и способны творить чудеса, на которые не способны обычные программисты.)

1. Деструктору передается булевский признак: "Вызывать/не вызывать оператор delete после разрушения объекта". Деструктор класса Base виртуален, поэтому в нашем примере оператор delete p относится к правильному объекту класса Derived. В этом случае размер объекта известен уже на этапе компиляции — он равен sizeof(Derived) — и компилятор просто передает эту константу оператору delete.
2. Деструктор возвращает размер объекта. Мы можем сделать так (ведь мы сами написали компилятор, не правда ли?), чтобы каждый деструктор после разрушения объекта возвращал величину sizeof(Class). Эта схема также вполне работоспособна, поскольку деструктор класса Base виртуален. После его вызова система поддержки выполнения программ вызовет оператор delete, передавая ему результат работы деструктора.
3. Реализуется скрытая виртуальная функция-член, получающая размер объекта в качестве аргумента. Назовем ее _Size(), например. В этом случае система поддержки выполнения программ вызовет эту функцию, сохранит результат ее работы, уничтожит объект и вызовет оператор delete. Такая реализация может показаться неэффективной, но ее преимущество заключается в том, что компилятор может использовать функцию _Size() и для других целей.
4. Размер непосредственно хранится где-то в таблице виртуальных функций каждого класса. Это решение и гибко, и эффективно, но его нелегко реализовать.

(Окончен бал, погасли свечи, и мы из разработчиков компилятора разжалованы в рядовые программисты.) Как видим, для того чтобы передать оператору delete правильный размер блока, компилятор должен выполнить довольно много работы. Зачем же нам терять эту информацию и выполнять при каждом удалении объекта поиск, расходуя время?

Ведь все так хорошо складывается! Объекту класса SmallAllocator нужен размер удаляемого блока. Компилятор передает ему этот размер, а объект класса SmallObject переадресовывает его объекту класса FixedAllocator.

Большинство из перечисленных выше решений принято в предположении, что в классе Base определен виртуальный деструктор. Это лишний раз доказывает, насколько важно делать деструкторы полиморфных классов виртуальными. Если этим требованием пренебречь, удаление указателя типа Base, который на самом деле ссылается на объект производного класса, может привести к непредсказуемым результатам. В этом случае механизм распределения памяти в режиме отладки приведет к срабатыванию макросов assert, а в режиме NDEBUG просто вызовет крах программы. Каждый из нас согласится, что такое поведение можно считать "непредсказуемым".

Для того чтобы не заботиться об этом (и не проводить бессонные ночи, отлаживая программу, если вы вдруг забудете о том, что сказано выше), в классе SmallObject определен виртуальный деструктор. Любой класс, производный от класса Base, наследует его виртуальный деструктор. Это приводит нас к реализации класса SmallObject.

Во всем приложении нам нужен один-единственный объект класса SmallObjAllocator. Этот объект должен быть правильно создан и правильно разрушен, что само по себе трудно. К счастью, библиотека Loki позволяет решить эту проблему с помощью шаблонного класса SingletonHolder, описанного в главе 6. (Конечно, отсылать вас к следующим главам досадно, но еще досаднее потерять возможность повторного использования кода.) Пока можно рассматривать класс SingletonHolder как механизм, позволяющий осуществлять управление единственным экземпляром класса. Если этот класс имеет имя X, шаблонный класс конкретизируется как Singleton<X>. Затем, для того чтобы получить доступ к единственному экземпляру этого класса, нужно вызвать функцию Singleton<X>::Instance(). Шаблон проектирования Singleton (Одиночка) описан в книге Gamma et al. (1995).

Использование класса SingletonHolder позволяет чрезвычайно просто реализовать класс SmallObject.

```
typedef Singleton<SmallObjAllocator> MyAlloc;
void* SmallObject::operator new(std::size_t size)
{
   return MyAlloc::Instance().Allocate(size);
}
void SmallObject::operator delete(void* p, std::size_t size)
{
   MyAlloc::Instance().Deallocate(p, size);
}
```

4.8. Просто, сложно и снова просто

Реализация класса SmallObject оказалась довольно простой. Однако на самом деле не все так просто — ведь остались нерешенными проблемы, связанные с многопоточностью. Единственный объект класса SmallAllocator используется всеми экземплярами класса SmallObject. Если эти экземпляры принадлежат разным потокам, то объект класса SmallObjAllocator придется распределять между ними. Как указано в приложении, в этом случае нужно предпринимать специальные меры. Кажется, нам придется пройтись по всем уровням нашей архитектуры, выделить критические операции и добавить соответствующую блокировку.

Однако многопоточность не является неразрешимой проблемой, поскольку в библиотеке Loki уже определены механизмы синхронизации объектов высокого уровня. Следуя поговорке "лучший способ повторного использования кода — его применение", включим заголовочный файл Threads.h из библиотеки Loki и внесем в класс SmallObject следующие изменения (они выделены полужирным шрифтом).

```
template <template <class T> class ThreadingModel>
class SmallObject : public ThreadingModel<SmallObject>
{
   ... как и раньше ...
};
```

Определения операторов new и delete также подвергаются пластической операции.

```
template <template <class T> class ThreadingModel>
void* SmallObject<ThreadingModel>::operator new(std::size_t size)
{
   Lock lock;
   return MyAlloc::Instance().Allocate(size);
}

template <template <class T> class ThreadingModel>
void SmallObject<ThreadingModel>::operator delete(void* p,
   std::size_t size)
{
   Lock lock;
   MyAlloc::Instance().Deallocate(p, size);
}
```

Вот и все! Все нижележащие уровни нашей реализации изменять не нужно — их функциональные возможности уже защищены блокировкой на верхнем уровне.

Синглтоны и механизмы многопоточности, предусмотренные в библиотеке Loki, свидетельствуют о могуществе повторного использования кода. Каждый из этих вопросов — время жизни глобальной переменной и многопоточность — по-своему сложен. Преодолеть эти сложности на основе первого способа реализации класса SmallObject слишком трудно — хотя бы потому, что в классе FixedAllocator приходится выполнять кэширование при инициализации одного и того же объекта для каждого потока в отдельности.

4.9. Применение

В этом разделе показано, как использовать файл SmallObj.h в приложениях.

Для того чтобы применить класс SmallObj, нужно задать соответствующие параметры конструктора класса SmallAllocator: размер объекта класса Chunk и максимальный размер объекта, который может еще считаться небольшим. Что значит "небольшой" объект? Какие размеры считаются "небольшими"?

Чтобы ответить на эти вопросы, вернемся к предназначению механизма распределения памяти для небольших объектов. С его помощью мы хотели повысить эффективность использования памяти и понизить затраты времени, преодолев ограничения, наложенные стандартным механизмом.

Размеры дополнительной памяти, которая используется стандартным распределителем, сильно варьируются. Как-никак, стандартный механизм распределения памяти тоже может применять стратегии, изложенные в этой главе. В большинстве случае, однако, следует ожидать, что размер вспомогательной памяти на обычных машинах будет изменяться от 4 до 32 байт для каждого объекта. Для распределителя памяти, который для каждого объекта резервирует 16 байт дополнительной памяти, непроизводительные затраты составят 25 %; таким образом, объект размером 64 байт можно считать "небольшим".

С другой стороны, если объект класса SmallAllocator размещает в памяти большие объекты, в конце концов выделенной памяти окажется намного больше, чем

нужно (не забывайте, что класс FixedAllocator стремится оставить в памяти хотя бы один объект класса Chunk, даже если все объекты удалены).

Библиотека Loki предоставляет выбор. В файле SmallObj.h определены три символа препроцессора, приведенные в табл. 4.1. Все исходные файлы, входящие в проект, следует компилировать, указав один и тот же символ препроцессора (или не задавая его вообще). Если этого не сделать, ничего смертельного не случится — просто будет создано больше объектов FixedAllocator, предназначенных для разных размеров.

Параметры, предусмотренные по умолчанию, предназначены для машин с разумным объемом физической памяти. Если в директиве #define символы MAX_SMALL_OBJECT_SIZE или DEFAULT_CHUNK_SIZE определить равными нулю, то в заголовочном файле SmallObj.h будет применяться условная компиляция, которая просто использует обычные операторы ::operator new и ::operator delete, не прибегая к выделению вспомогательной памяти вообще. Интерфейс объектов остается прежним, но их функции становятся подставляемыми заглушками (inline stubs), что приводит к стандартному механизму распределения динамической памяти.

Обычно шаблонный класс SmallObject имеет только один параметр. Для поддержки разных размеров участков памяти и небольших объектов этот класс получает еще два шаблонных параметра. По умолчанию ими являются константы DEFAULT_CHUNK_SIZE и MAX_SMALL_OBJECT_SIZE соответственно.

```
template
<
   template <class T>
      class ThreadingModel = DEFAULT_THREADING,
   std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
   std::size_t maxSmallObjectSize = NAX_SMALL_OBJECT_SIZE
>
class SmallObject;
```

Если просто написать SmallObj<>, вы получите класс, который может работать со стандартной моделью потоков.

Таблица 4.1. Символы препроцессора, использованные в файле SmallObj.h

Символ	Предназначение	Значение по умолчанию
DEFAULT_CHUNK_SIZE	Размер участка памяти (в байтах), заданный по умолчанию	4096
MAX_SMALL_OBJECT_SIZE	Максимальное значение, обрабатываемое классом SmallObjAllocator	64
DEFAULT_THREADING	Стандартная модель потоков, используемая в приложении. В многопоточном приложении этот символ следует определять как ClassLevelLockable	Наследуется от файла Threads.h

4.10. Резюме

В некоторых идиомах языка C++ очень широко применяются небольшие объекты, расположенные в динамической памяти. Это происходит благодаря тому, что в языке C++ динамический полиморфизм (runtime polymorphism) тесно связан с распределением динамической памяти и семантикой указателей и ссылок. Однако стандартные

механизмы распределения памяти (с помощью операторов ::operator new и ::operator delete) часто оптимизируются для работы с большими объектами, а не маленькими. Это делает стандартный механизм распределения памяти непригодным для работы с небольшими объектами, поскольку он медленно работает и неэффективно использует память.

Для решения этой проблемы необходимо создать специальные механизмы распределения памяти для небольших объектов, настроенные на работу с маленькими блоками памяти (от десятков до сотен байтов). Эти механизмы объединяют элементарные блоки в более крупные участки (chunks), стремясь минимизировать накладные расходы памяти и времени. Система поддержки выполнения программ на языке C++ сама определяет размер освобождаемого блока. Эту информацию можно перехватить, просто применяя малоизвестную форму перегрузки оператора delete.

Может ли механизм распределения памяти для небольших объектов, предусмотренный в библиотеке Loki, работать еще быстрее? Конечно, да. Этот механизм не выходит за пределы стандарта языка C++. Как показано выше, такие проблемы, как выравнивание блоков памяти, решаются традиционным способом, что приводит к снижению эффективности его работы. И все же этот механизм работает достаточно быстро, просто и надежно, что гарантирует его платформную независимость.

4.11. Краткое описание механизма распределения памяти для небольших объектов

- Механизм, реализованный в библиотеке Loki, имеет четыре уровня. Первый уровень состоит из закрытого типа Chunk, предназначенного для организации памяти в виде участков (chunks), состоящих из блоков одинакового размера. На втором уровне архитектуры находится класс FixedAllocator, в котором используется вектор переменной длины, состояший из участков памяти. Этот класс предназначен для выполнения запросов на выделение динамической памяти. На третьем уровне расположен класс SmallObjAllocator, использующий несколько объектов класса FixedAllocator. Это позволяет выделять память для объекта любого размера. Небольшие объекты размещаются в памяти с помощью объекта класса FixedAllocator, а запросы на выделение больших участков памяти переадресовываются стандартному оператору ::operator new. Последний, четвертый, уровень содержит шаблонный класс SmallObject, реализующий интерфейс объектов класса SmallObjAllocator.
- Краткое описание шаблонного класса SmallObject выглядит так.

```
template
<
   template <class T>
   class ThreadingModel = DEFAULT_THREADING,
   std::size_t chunkSize = DEFAULT_CHUNK_SIZE,
   std::size_t maxSmallObjectSize = MAX_SMALL_OBJECT_SIZE
>
class SmallObject
{
 public:
   static void* operator new(std::size_t size);
   static void operator delete(void* p, std::size_t size);
   virtual ~SmallObject()
```

- Функциональные возможности механизма распределения памяти для небольших объектов можно унаследовать от конкретизации класса SmallObject. Шаблонный класс SmallObject можно конкретизировать с параметрами, заданными по умолчанию (SmallObject<>), настроить его модель потоков или задать параметры выделяемой памяти.
- Если в нескольких потоках объекты создаются с помощью оператора new, следует использовать многопоточную модель, задав параметр ThreadingModel. Информация об этой модели приведена в приложении.
- Значение константы DEFAULT_CHUNK_SIZE по умолчанию равно 4096.
- Значение константы MAX_SMALL_OBJECT_SIZE по умолчанию равно 64.
- В директиве #define можно определить константы MAX_SMALL_OBJECT_SIZE или DEFAULT_CHUNK_SIZE, или обе одновременно, заместив их значения, принятые по умолчанию. После расширения макрос распространяется на константы типа std::size_t (или совместимого типа).
- Если в директиве #define константы MAX_SMALL_OBJECT_SIZE или DEFAULT_CHUNK_SIZE определить равными нулю, то в заголовочном файле SmallObj.h будет применяться условная компиляция, которая просто использует стандартный механизм распределения памяти. Интерфейс остается прежним. Это полезно, если нужно сравнить работу программы со специализированным механизмом распределения памяти и без него.

Часть II

Компоненты

5

Обобщенные функторы

Эта глава посвящена обобщенным функторам (generalized functors) — мощной абстракции, позволяющей осуществлять несвязное взаимодействие между объектами (decoupled interobject communication). Обобщенные функторы особенно полезны, когда в объектах необходимо хранить запросы. Шаблон проектирования, описывающий инкапсулированные запросы, называется `Command` (Gamma et al., 1995).

Кратко говоря, обобщенный функтор — это *любой вызов процедуры, позволенный в языке С++, инкапсулированный в объекте первого класса, гарантирующем типовую безопасность* (typesafe first-class object). (К первому классу относятся типы, обладающие всеми свойствами встроенных типов. Например, в языке С++ типы `int` и `char` являются типами первого класса. Объекты первого класса обладают полной семантикой значений. Их можно объявлять в любом месте программы, присваивать любым переменным, копировать и передавать по ссылке или по значению, а также возвращать в качестве значения функции. — *Прим. ред.*)

Более детальное определение приводится ниже.

- Обобщенный функтор *инкапсулирует* вызов любой процедуры, поскольку он получает указатели на простые функции, функции-члены, функторы и даже на другие обобщенные функторы, а также некоторые или все их аргументы.
- Обобщенный функтор гарантирует *типовую безопасность*, поскольку он никогда не передает аргументы неверного типа неправильным функциям.
- Обобщенный функтор является *объектом, имеющим семантику значений*, так как он полностью поддерживает копирование, присваивание и передачу параметров по значению. Обобщенный функтор можно свободно копировать, причем он не может содержать виртуальные функции-члены.

Обобщенные функторы позволяют хранить вызовы процедур в виде значений, передавать их как параметры и выполнять далеко от места их создания. Они представляют собой усовершенствованный вариант указателей на функции. Существенное различие между указателями на функции и обобщенными функторами заключается в том, что функторы могут хранить состояние объекта и вызывать его функции-члены.

В этой главе изложены следующие сведения.

- Что такое шаблон проектирования `Command` и как он связан с обобщенными функторами.
- В каких ситуациях они могут принести пользу.
- Внутреннее устройство различных функциональных сущностей в языке С++ и способы их инкапсуляции с помощью единообразного интерфейса.

- Как хранить вызов процедуры и некоторые или все ее аргументы внутри объекта, передавать ее другим объектам и свободно ее вызывать.
- Как связывать между собой несколько отложенных вызовов и последовательно их выполнять.
- Как использовать мощный шаблонный класс `Functor`, реализующий описанные выше функциональные возможности.

5.1. Шаблон Command

В классической книге о шаблонах проектирования (Gamma et al., 1995) утверждается, что шаблон `Command` предназначен для *инкапсулирования запроса внутри объекта*. Объект этого класса представляет собой часть работы, хранящуюся отдельно от своего исполнителя. Общая структура шаблона `Command` представлена на рис. 5.1.

Основной частью шаблона является сам класс `Command`. Его основное предназначение — ослабить зависимость между двумя частями системы: вызывающим модулем и получателем.

Типичная последовательность действий такова.

1. Приложение (клиент) создает объект класса `ConcreteCommand`, передавая ему информацию, которой достаточно для выполнения задачи. Пунктирная линия на рис. 5.1 иллюстрирует тот факт, что клиент влияет на состояние объекта класса `ConcreteCommand`.
2. Приложение передает интерфейс `Command` объекта класса `ConcreteCommand` вызывающему объекту, который сохраняет этот интерфейс.
3. Позднее вызывающий объект выбирает момент для начала действий и активизирует виртуальную функцию-член `Execute` из класса `Command`. Механизм виртуальных вызовов переадресует этот вызов объекту класса `ConcreteCommand`, который отслеживает все детали. Объект класса `ConcreteCommand` связывается с объектом класса `Receiver` (это одно из его заданий) и использует его для выполнения реальной обработки данных, например, вызывая функцию-член `Action`. В противном случае объект класса `ConcreteCommand` может сам выполнить всю работу. В этом случае объект класса `Receiver` на рис. 5.1 не нужен.

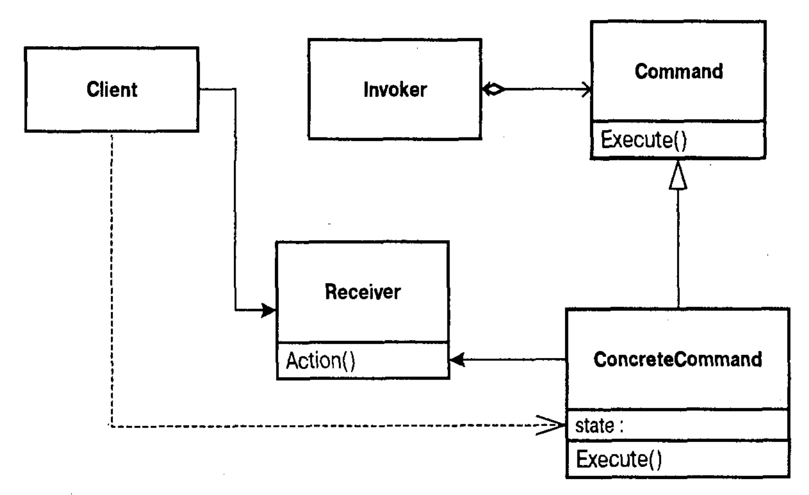


Рис. 5.1. Шаблон проектирования Command

Вызывающий объект может свободно вызывать функцию Execute. Еще более важно то, что в ходе выполнения программы в вызывающий объект можно встроить разные действия, заменив объект класса Command, который в нем хранится.

Следует отметить два момента. Во-первых, вызывающий модуль не знает, как выполняется работа. Это отнюдь не ново — чтобы использовать алгоритм сортировки, не обязательно знать, как именно он реализован. Однако вызывающий объект не знает даже, для какого вида обработки предназначен класс Command. (Однако алгоритм сортировки обязан выполнять именно сортировку, а не что-то иное.) Вызывающий объект лишь вызывает функцию Execute из объекта класса Command при наступлении определенного события. С другой стороны, получатель не обязан знать, что его функция-член Action была вызвана каким-то объектом.

Таким образом, объект класса Command гарантирует изоляцию вызывающего объекта от получателя вызова. Они могут быть абсолютно взаимно невидимыми, общаясь лишь через объект класса Command. Обычно связи между вызывающим объектом и получателем вызова устанавливает объект класса Application. Это значит, что для заданного множества получателей можно использовать разные вызывающие модули, причем в заданный вызывающий модуль можно встраивать разных получателей — и все это происходит в условиях полной изоляции этих объектов.

Во-вторых, посмотрите на шаблон Command в перспективе. В обычных программистских задачах, когда нужно выполнить какое-то действие, при вызове задаются объект, его функция-член и ее аргументы.

```
window.Resize(0, 0, 200, 100); // Изменение размеров окна
```

Момент инициализации такого вызова в принципе невозможно отличить от момента сбора его элементов в одно целое (объекта, процедуры и аргументов). Однако в шаблоне Command вызывающий объект уже содержит элементы вызова, откладывая сам вызов на неопределенное время. Это иллюстрируется следующим примером.

```
Command resizeCmd(
   window,              // Объект
   &Window::Resize,     // Функция-член
   0, 0, 200, 100);     // Аргументы
// Позднее ...
resizeCmd.Execute();    // Изменение размера окна
```

(Ниже мы остановимся на малоизвестной конструкции языка C++ &Window::Resize.) В шаблоне Command момент *сбора* информации об окружении, необходимой для обработки данных, отличается от момента выполнения собственно *обработки*. В промежутке между этими двумя моментами программа хранит и передает запрос на обработку в виде объекта. Если бы не существовала возможность отложить вызов, не было бы и самого шаблона Command. С этой точки зрения само существование объекта класса Command обусловлено возможностью откладывания вызова, поскольку запрос в это время нужно где-то хранить.

Отсюда следуют два важных аспекта шаблона Command.

- *Изоляция интерфейсов*. Вызывающий объект изолирован от получателя вызова.
- *Разделение времени*. Объект класса Command хранит готовый к выполнению отложенный запрос.

Знание окружения также необходимо. *Окружение* (environment) точки выполнения — это множество сущностей (переменных и функций), которые являются видимыми из этой точки. В момент начала обработки необходимое окружение должно

быть доступным, в противном случае запуск невозможен. Объект класса ConcreteCommand может хранить часть нужной информации о своем окружении в виде своего состояния, а доступ к остальной информации получать во время вызова функции Execute. Чем больше информации об окружении хранится в объекте класса ConcreteCommand, тем более он независим.

С точки зрения реализации можно идентифицировать две разновидности классов ConcreteCommand. Некоторые из них просто поручают работу получателю, вызывая функцию-член объекта класса Receiver. Такие классы называются *командами пересылки* (forwarding commands). Другие классы выполняют более сложную работу. Они также могут вызывать функции-члены других объектов, но имеют более сложную логику функционирования. Такие классы называются *активными командами* (active commands).

Такая классификация команд позволяет очертить границы обобщенной реализации. Активные команды нельзя описать раз и навсегда — их код по определению зависит от конкретного приложения. В то же время на основе команд пересылки можно создавать шаблонные классы. Поскольку команды пересылки напоминают указатели на функции и похожи на функторы, мы будем называть их *обобщенными функторами* (generalized functors).

Оставшаяся часть главы посвящена разработке шаблонного класса Functor, инкапсулирующего любой объект, любую функцию-член этого объекта и любой набор аргументов, относящихся к этой функции. В момент активизации обобщенный функтор собирает все компоненты в одно целое, создавая вызов функции.

Объект класса Functor может оказать неоценимую помощь при проектировании с использованием шаблона Command. В реализациях, разработанных вручную, шаблон Command масштабируется не слишком хорошо — приходится писать много маленьких классов Command (по одному на каждую операцию: CmdAddUser, CmdDeleteUser, CmdModifyUser и т.д.), каждый из которых содержит тривиальную функцию-член Execute, просто вызывающую конкретную функцию-член некоторого объекта. Класс Functor, обладающий возможностью вызывать любую функцию-член любого объекта, мог бы оказать в этом деле неоценимую помощь.

В классе Functor стоит реализовать и некоторые из активных команд, например, упорядочение нескольких последовательных операций. Тогда объект класса Fuctor мог бы собирать несколько операций и выполнять их по порядку. В книге Gamma et al. (1995) такие полезные объекты описаны под именем MacroCommand.

5.2. Шаблон Command в реальном мире

Применение шаблона Command часто иллюстрируют на примере разработки оконного интерфейса. В хороших объектно-ориентированных специализированных системах для разработки графического пользовательского интерфейса шаблон Command в той или иной форме применяется уже многие годы.

Создателям оконных интерфейсов необходим стандартный способ передачи пользовательских действий (например, щелчков мыши или нажатий клавиш) приложению. Когда пользователь щелкает на кнопке, выбирает пункт меню или делает что-нибудь подобное, оконная система должна уведомить об этом соответствующее приложение. С точки зрения оконной системы команда Options в меню Tools не имеет никакого особого смысла. Это накладывает на приложение очень жесткие ограничения. Объект класса Command позволяет эффективно изолировать приложение от оконного интерфейса, выступая в роли посредника, сообщающего приложению о действиях пользователя.

Например, в оконной системе в роли вызывающих объектов выступают элементы интерфейса (кнопки, пункты меню и т.п.), а получателем является объект, определяющий реакцию приложения на действия пользователя (например, диалоговое окно или само приложение).

Объекты класса Command представляют собой средство общения между пользовательским интерфейсом и приложением. Как указывалось в предыдущем разделе, класс Command обеспечивает удвоенную гибкость. Во-первых, в оконную систему можно встраивать новые элементы пользовательского интерфейса, не изменяя логику работы приложения. В таких случаях говорят, что программа имеет *оболочку* (skinnable), поскольку новые элементы интерфейса можно добавлять, не касаясь самого приложения. Оболочки не имеют никакой архитектуры — они только предоставляют места для объектов класса Command и знают, как с ними взаимодействовать. Во-вторых, одни и те же элементы пользовательского интерфейса можно повторно использовать в разных приложениях.

5.3. Вызываемые сущности в языке C++

Для того чтобы создать обобщенную реализацию команд пересылки, попытаемся описать связанные с ними понятия в терминах языка C++.

Команда пересылки представляет собой *обобщенный обратный вызов* (generalized callback). Обратный вызов — это указатель на функцию, который можно передавать и вызывать в любое время, как показано в следующем примере.

```
void Foo();
void Bar();

int main()
{
   // Определяем указатель на функцию,не имеющую
   // параметров и возвращающую значение типа void.
   // Инициализируем этот указатель адресом функции Foo
   void (*pF)() = &Foo;
   Foo();               // Непосредственный вызов функции Foo;
   Bar();               // Непосредственный вызов функции Bar;
   (*pF)();             // Вызов функции Foo через указатель pF
   void (*pF2)() = pF;  // Создаем копию указателя pF
   pF = &Bar;           // Устанавливаем указатель pF
                        // на функцию Bar
   (*pF)();             // Вызов функции Bar через указатель pF
   (*pF2)();            // Вызов функции Foo через указатель pF2
}
```

Между непосредственным вызовом функции Foo и вызовом через указатель (*pF)[1] есть одно существенное отличие. Во втором случае указатель можно копировать, где-то хранить и устанавливать на другую функцию, вызывая ее при необходимости. Следовательно, указатель на функцию очень похож на команду пересылки, в которой значительная часть работы хранится отдельно от объекта, выполняющего фактическую обработку данных.

---

[1] Компилятор предлагает синтаксическое сокращение: выражение (*pF)() эквивалентно pF(). Однако выражение (*pF)() более наглядно отражает суть происходящего — указатель pF разыменовывается и к нему применяется оператор вызова функции ().

Действительно, обратный вызов — это способ использования шаблона Command в языке С. Например, система X Windows хранит такой обратный вызов для каждого пункта меню и каждого элемента интерфейса (widget). Когда пользователь выполняет определенные действия (например, щелкает на кнопке), соответствующий компонент интерфейса активизирует обратный вызов, причем компонент не знает, что при этом происходит.

Кроме простых обратных вызовов, в языке C++ есть еще много сущностей, поддерживающих оператор().

- Функции.
- Указатели на функции.
- Ссылки на функции (по существу, представляющие собой константные указатели на функции).
- Функторы, т.е. объекты, в которых определен оператор ().
- Результат применения операторов .* и ->* к указателям на функции-члены.

К каждой из указанных сущностей можно приписать пару круглых скобок, задать внутри них набор аргументов и выполнить какую-нибудь обработку данных. С другими сущностями в языке C++, кроме перечисленных выше, этого делать нельзя.

Сущности, позволяющие применять оператор (), называются ***вызываемыми*** (callable). Цель этой главы — реализовать набор команд пересылки, способных хранить и передавать вызов любой вызываемой сущности[2]. Шаблонный класс Functor инкапсулирует команды пересылки и обеспечивает единообразный интерфейс.

Реализация должна учитывать три основных варианта: простые вызовы функций, вызовы функторов (включая вызовы объектов класса Functor, т.е. возможность передавать вызовы от одного объекта класса Functor другому) и вызовы функций-членов. Для этого следует создать абстрактный базовый класс и подкласс для каждого класса. На первый взгляд все это можно сделать с помощью средств языка C++. Однако, как только вы приступите к разработке программы, на вас обрушится ворох проблем.

5.4. Скелет шаблонного класса Functor

Для класса Functor обязательно следует проверить идиому "дескриптор-тело" ("handle-body"), впервые предложенную в работе (Coplien, 1992). В главе 7 подробно объясняется, что в языке C++ голый указатель на полиморфный тип не имеет полноценной семантики из-за проблем, связанных с его принадлежностью. Для того чтобы снять ответственность за управление временем жизни объектов с клиентов класса Functor, лучше всего наделить класс Functor семантикой значений (хорошо определенными операциями копирования и присваивания). Класс Functor имеет полиморфную реализацию, но этот факт скрыт внутри него. Назовем реализацию базового класса именем FunctorImpl.

Отметим важную особенность: функция Command::Execute шаблона Command в языке C++ перевоплощается в оператор (), определенный пользователем. Это еще один аргумент в пользу применения оператора (). Для программистов на языке C++

---

[2] Обратите внимание на то, что мы ничего не говорим о типах. Мы могли бы просто сказать: "Типы, допускающие выполнение оператора (), называются вызываемыми сущностями". Однако, хотя это кажется невероятным, в языке C++ есть сущности, не имеющие типа и в то же время позволяющие применять оператор ().

оператор вызова функции имеет точный смысл — “выполнить”. Однако намного важнее то, что этот оператор обеспечивает синтаксическое единообразие. Класс Functor не только передает вызов вызываемой сущности, он и сам является таковой. В подобном случае объект класса Functor может содержать другие объекты этого класса. С этого момента мы будем считать класс Functor частью множества вызываемых сущностей. Это позволит нам многие вещи делать единообразно.

Проблемы возникнут, как только мы попытаемся определить оболочку класса Functor. Вначале она может выглядеть следующим образом.

```
class Functor
{
public:
   void operator()();
   // другие функции-члены
private:
   // реализация класса
};
```

Первый вопрос: какой тип должно иметь значение, возвращаемое оператором ()? Должно ли оно иметь тип void? В некоторых случаях может понадобиться вернуть что-то еще, например, значение типа bool или std::string. Нет никаких причин отказываться от возврата параметризованных значений.

Для решения таких проблем предназначены шаблонные классы, позволяющие избежать лишних хлопот.

```
template <typename ResultType>
class Functor
{
public:
   ResultType operator()();
   // Другие функции-члены
private:
   // Реализация
};
```

Это решение выглядит вполне приемлемо, хотя теперь у нас уже не один класс Functor, а целое семейство. Это вполне разумно, поскольку функторы, возвращающие строки, и функторы, возвращающие целые числа, имеют разные функциональные возможности.

Второй вопрос: должен ли оператор (), принадлежащий функтору, также получать аргументы? Может возникнуть необходимость передать объекту класса Functor некую информацию, которая была недоступна в момент его создания. Например, если щелчки мышью в окне пересылаются через функтор, вызывающий объект передает информацию о координатах окна (известную только в момент вызова) объекту класса Functor, вызывая его оператор ().

Более того, в обобщенном программировании количество параметров не ограничивается, причем они могут иметь любой тип. Таким образом, нет никаких причин накладывать ограничения ни на количество, ни на тип передаваемых функтору параметров.

Отсюда следует, что каждый функтор определяется типом возвращаемого им значения и типами его аргументов. Для этого понадобится мощная языковая поддержка: переменные шаблонные параметры в сочетании с переменными параметрами вызова функций.

К сожалению, переменных шаблонных параметров в языке C++ просто нет. Вместо них предусмотрены функции с переменными аргументами (так же, как и в языке C). В языке C с их помощью выполняется значительная часть работы (при условии, что вы

очень осторожны), но для языка C++ этого недостаточно. Переменные аргументы поддерживаются с помощью эллипсисов (таких как `printf` или `scanf`). Вызов функций `printf` или `scanf`, когда спецификация формата не соответствует количеству и типу их аргументов, — весьма распространенная и опасная ошибка, иллюстрирующая недостатки эллипсисов. Механизм переменных параметров ненадежен, относится к низкому уровню и не соответствует объектной модели языка C++. Короче говоря, используя эллипсисы, вы остаетесь без типовой безопасности, объектной семантики (применение объектов, содержащих эллипсисы, приводит к непредсказуемым последствиям) и ссылочных типов. Для вызываемых функций недоступно даже количество их аргументов. Действительно, там, где есть эллипсисы, языку C++ делать нечего.

В качестве альтернативы можно ограничить количество аргументов функтора произвольным достаточно большим числом. Такая неопределенность — одна из самых неприятных проблем для программиста. Однако этот выбор можно сделать на основе экспериментальных наблюдений. В библиотеках (особенно старых) количество параметров функций не превышает 12. Установим предельное количество параметров равным 15 и забудем об этой неприятности.

Даже это не облегчило нашей участи. В языке C++ не допускается применение шаблонов, имеющих одинаковые имена, но разное количество параметров. Это значит, что приведенный ниже фрагмент программы является неправильным.

```
// Функтор без аргументов
template <typename ResultType>
class Functor
{
  ...
};

// Функтор с одним аргументом
template <typename ResultType, typename Parm1>
class Functor
{
  ...
};
```

Называть шаблонные классы именами `Functor1`, `Functor2` и так далее может оказаться затруднительно.

В главе 3 описаны списки типов, позволяющие работать с коллекциями типов. Типы параметров функтора также образуют коллекцию, поэтому списки типов подойдут нам идеально. Определение шаблонного класса `Functor` с помощью списков типов приведено ниже.

```
// Функтор, имеющий любое количество аргументов любого типа
template <typename ResultName, class TList>
class Functor
{
  ...
}
```

Возможная конкретизация этого класса выглядит следующим образом.

```
// Определяем функтор, получающий параметры типа int и double
// и возвращающий значение типа double
Functor<double, TYPELIST_2(int, double)> myFunctor;
```

Одним из достоинств этого подхода является возможность повторно использовать сущности, определенные списками типов, а не разрабатывать новые.

Как мы вскоре убедимся, списки типов хотя и полезны, но не решают всех проблем. Мы по-прежнему вынуждены создавать отдельную реализацию функтора для каждого конкретного количества аргументов. В дальнейшем для простоты ограничимся двумя аргументами. Масштабировать программу для конкретного количества параметров (не больше 15) можно с помощью включения заголовочного файла `Functor.h`.

Полиморфный класс `FunctorIml`, погруженный в класс `Functor`, имеет те же самые шаблонные параметры.[3]

```
template <typename R, class TList>
class FunctorImpl;
```

Класс `FunctorImpl` определяет полиморфный интерфейс, абстрагирующий вызов функции. Для каждого конкретного количества параметров определена явная специализация класса `FunctorImpl` (глава 2). В каждой специализации определена чисто виртуальная функция `()` для определенного количества и типов параметров.

```
template <typename R>
class FunctorImpl<R, NullType>
{
public:
   virtual R operator()() = 0;
   virtual FunctorImpl* Clone() const = 0;
   virtual ~FunctorImpl() {}
};

template <typename R, typename P1>
class FunctorImpl<R, TYPELIST_1(P1)>
{
public:
   virtual R operator()(P1) = 0;
   virtual FunctorImpl* Clone() const = 0;
   virtual ~FunctorImpl() {}
};

template <typename R, typename P1, typename P2>
class FunctorImpl<R, TYPELIST_2(P1, P2)>
{
public:
   virtual R operator()(P1, P2) = 0;
   virtual FunctorImpl* Clone() const = 0;
   virtual ~FunctorImpl() {}
};
```

Классы `FunctorImpl` представляют собой частичные специализации исходного шаблонного класса `FunctorImpl`. Их свойства подробно описаны в главе 2. В нашем случае частичная шаблонная специализация позволяет определять разные версии класса `FunctorImpl` в зависимости от количества элементов в списке типов.

Кроме оператора `()` в классе `FunctorImpl` определены две вспомогательные функции-члены — `Clone` и виртуальный деструктор. Функция `Clone` предназначена для создания полиморфной копии объекта класса `FunctorImpl`. (Детали полиморфного

---

[3] Использование ключевых слов `typename` или `class` для определения шаблонных параметров приводит к эквивалентным результатам. В книге по умолчанию при определении шаблонных параметров, которые могут относиться к элементарным типам (например `int`), принято использовать ключевое слово `typename`, а ключевое слово `class` применяется для шаблонных параметров, тип которых определяется пользователем.

клонирования изложены в главе 8.) Виртуальный деструктор позволяет уничтожать объекты классов, производных от класса FunctorImpl, применяя оператор delete к указателю на объект класса FunctorImpl. Важность этого деструктора подробно обсуждалась в главе 4.

Классическая реализация класса Functor приведена ниже.

```
template <typename R, class TList>
class Functor
{
public:
   Functor();
   Functor(const Functor&);
   Functor& operator=(const Functor&);
   explicit Functor(std::auto_ptr<Impl> spImpl);
   ...
private:
   // Определение тела функтора
   typedef FunctorImpl<R, TList> Impl;
   std::auto_ptr<Impl> spImpl_;
};
```

В классе Functor интеллектуальный указатель на класс FunctorImpl<R, TList>, представляющий собой соответствующий тип тела функтора, хранится в виде закрытого члена. Для этого выбран стандартный интеллектуальный указатель std::auto_ptr.

Приведенный выше код иллюстрирует наличие у класса Functor определенных артефактов, демонстрирующих его семантику значений. К этим артефактам относятся конструктор по умолчанию, конструктор копирования и оператор присваивания. Явный деструктор не нужен, поскольку интеллектуальный указатель auto_ptr автоматически освобождает все ресурсы.

Кроме того, в классе Functor определен "конструктор расширения", предоставляющий классу FunctorImpl доступ к указателю auto_ptr. Конструктор расширения позволяет определять классы, производные от класса FunctorImpl, и непосредственно инициализировать класс Functor указателями на них.

Почему аргументом конструктора расширения является итератор auto_ptr, а не обычный указатель? Создание объекта с помощью указателя auto_prt явно свидетельствует о том, что объект класса FunctorImpl принадлежит классу Functor. Вызывая этот конструктор, пользователь класса Functor должен явно указать тип auto_ptr. Если он сделал это, значит, понимает, о чем идет речь.[4]

---

[4] Разумеется, это вовсе не обязательно. Однако это все же лучше, чем молча выбирать один вариант (копирование или владение). Хорошие библиотеки на языке C++ отличаются одной интересной особенностью: когда может возникнуть неопределенность, они позволяют пользователю устранять ее с помощью явного кода. С другой стороны, существуют библиотеки, неправильно использующие свойства языка C++, определенные по умолчанию (особенно преобразования типов и владение указателями). Они предоставляют пользователям меньше возможностей для дополнительного программирования, но в качестве компенсации принимают сомнительные предположения и решения, облегчая работу пользователя.

5.5. Реализация оператора пересылки Functor::operator()

В классе Functor необходим оператор (), который пересылал бы вызов оператору FunctorImpl::operator(). Для этого можно было бы воспользоваться подходом, который мы применяли при разработке самого класса FunctorImpl, и создать группу частичных шаблонных специализаций, каждая из которых соответствует конкретному количеству параметров. Однако этот подход здесь не годится. В классе Functor содержится довольно большое количество кода, и было бы неразумной тратой времени и памяти размножать его только для того, чтобы реализовать оператор ().

Однако сначала попробуем определить типы параметров. Тут нам на помощь придут списки типов.

```
template <typename R, class TList>
class Functor
{
   typedef TList ParmList;
   typedef typename TypeAtNonStrict<TList, 0, EmptyType>::
      Result Parm1;
   typedef typename TypeAtNonStrict<TList, 1, EmptyType>::
      Result Parm2;
   ...  как и раньше ..
};
```

Класс TypeAtNonStrict является шаблоном, обладающим доступом к типу по его позиции в списке типов. Если тип не найден, в качестве третьего шаблонного аргумента класса TypeAtNonStrict задается результат (т.е. внутренний класс TypeAtNonStrict<...>::Result). В качестве третьего аргумента выбран класс EmptyType, являющийся, как следует из его названия, пустым. (Подробное описание класса TypeAtNonStrict можно найти в главе 3, а описание класса EmptyType — в главе 2.) Итак, тип Parm*N* будет либо *N*-м элементом списка типов, либо типом EmptyType, если количество элементов списка типов меньше, чем *N*.

Чтобы реализовать оператор (), воспользуемся интересным трюком. Определим все версии оператора () — для любого количества параметров — внутри определения класса Functor.

```
template <typename R, class TList>
class Functor
{
   ... как и раньше ...
public:
   R operator()()
   {
      return (*spImpl_)();
   }
   R operator()(Parm1 p1)
   {
      return (*spImpl_)(p1);
   }
   R operator()(Parm1 p1, Parm2 p2)
   {
      return (*spImpl_)(p1, p2);
   }
};
```

В чем состоит трюк? Для данной конкретизации класса Functor правильной является только одна версия оператора (). Все остальные версии на этапе компиляции порождают сообщения об ошибках. Можно предположить, что класс Functor не будет скомпилирован вовсе, поскольку в каждой специализации класса FunctorImpl определен только один оператор (), а не группа, как в классе Functor. Трюк заключается в том, что в языке C++ функции-члены шаблонных классов не конкретизируются, *пока они не вызваны*. Пока не вызывается неправильный оператор (), компилятору все равно. При вызове перегруженного оператора (), не имеющего смысла, компилятор попытается сгенерировать его тело и обнаружит несоответствие.

```
// Определяем функтор, получающий параметры типа int и double,
// и возвращающий значение типа double.
Functor<double, TYPELIST_2(int, double)> myFunctor;
// Вызываем его.
// Генерируется тело оператора operator()(double, int).
double result = myFunctor(4, 5.6);
// Неверный вызов.
double result = myFunctor(); //Ошибка!
// Оператор operator()() неверен,
// потому что он не определен в классе
// FunctorImpl<double, TYPELIST_2(int, double)>.
```

Благодаря этому тонкому трюку класс Functor не обязательно частично инициализировать для нуля, одного, двух и больше параметров — это лишь приведет к дублированию кода. Достаточно определить все версии и позволить компилятору генерировать только одну из них.

Теперь все вспомогательные средства созданы, и можно приступать к определению конкретных классов, производных от класса FunctorImpl.

5.6. Работа с функторами

Начнем работать с функторами. Функторы представляют собой экземпляры классов, в которых определен оператор (), как в классе Functor (т.е. объект класса Functor и сам является функтором). Следовательно, конструктор класса Functor, получающий объект класса Functor в качестве параметра, является шаблоном, параметризованным типом этого функтора.

```
template <typename R, class TList>
class Functor
{
   ... как и раньше ...
public:
   template <class Fun>
   Functor(const Fun& fun);
};
```

Для того чтобы реализовать этот конструктор, нам нужен простой шаблонный класс FunctorHandler, производный от класса FunctorImpl<R, TList>. Этот класс хранит объект типа Fun и передает ему оператор (). Реализуя оператор (), мы прибегнем к описанному выше трюку.

Чтобы избежать определения слишком большого количества параметров класса FunctorHandler, сделаем шаблонным параметром саму конкретизацию класса Functor. Этот единственный параметр содержит остальные, поскольку он предусматривает внутренний оператор typedef.

```
template <class ParentFunctor, typename Fun>
class FunctorHandler
   : public FunctorImpl
      <
         typename ParentFunctor::ResultType,
         typename ParentFunctor::ParmList
      >
{
public:
   typedef typename ParentFunctor::ResultType ResultType;

   FunctorHandler(const Fun& fun) : fun_(fun) {}
   FunctorHandler* Clone() const
   { return new FunctorHandler(*this); }

   ResultType operator()()
   {
      return fun_();
   }

   ResultType operator()(typename ParentFunctor::Parm1 p1)
   {
      return fun_(p1);
   }

   ResultType operator()(typename ParentFunctor::Parm1 p1,
      typename ParentFunctor::Parm2 p2)
   {
      return fun_(p1,p2);
   }
private:
   Fun fun_;
};
```

Класс FunctorHandler почти не отличается от класса Functor. Он переадресовывает запросы сохраняемой переменной-члену. Основное отличие заключается в том, что функтор хранится как значение, а не как указатель. Вот почему функторы являются неполиморфными типами с обычной семантикой копирования.

Обратите внимание на внутренние типы ParentFunctor::ResultType, ParentFunctor::Parm1 и ParentFunctor::Parm2. В классе FunctorHandler реализован простой конструктор, функция клонирования и несколько версий оператора (). Компилятор сам выберет правильный вариант. Если перегрузка оператора () выполнена неверно, компилятор выдаст сообщение об ошибке.

В реализации класса FunctorHadler нет ничего необычного, но это вовсе не означает, что она тривиальна. В следующем разделе будет показано, какой универсальностью обладает этот небольшой шаблонный класс.

С помощью объявления класса FunctorHandler легко написать шаблонный конструктор класса Functor.

```
template <typename R, class TList>
template <typename Fun>
Functor<R, TList>::Functor(const Fun& fun)
   : spImpl_(new FunctorHandler<Functor, Fun>(fun));
{
}
```

Здесь нет никакой ошибки. Два шаблонных параметра указаны правильно: выражение `template <class R, class TList>` относится к классу `Functor`, а `template <typename Fun>` является параметром конструктора. В стандарте языка такой код называется "шаблонным определением члена вне класса" ("out-of-class member template definition").

В теле конструктора переменная `spImpl_` устанавливается на новый объект типа `FunctorHandler`, конкретизированный и проинициализированный соответствующими аргументами.

Есть еще кое-что, о чем следовало бы упомянуть, кое-что, позволяющее лучше понять реализацию класса `FunctorHandler`. Обратите внимание на то, что при входе в тело конструктора класса `Functor` полная информация о типах уже содержится в шаблонном параметре `Fun`. При выходе из конструктора эта информация теряется, поскольку всем объектам класса `Functor` известен лишь указатель `spImpl_`, ссылающийся на базовый класс `FunctorImpl`. Эта очевидная потеря информации весьма примечательна: конструктор знает тип и действует подобно фабрике, преобразующей этот тип в полиморфное поведение. Информация о типах сохраняется в динамическом указателе на класс `FunctorImpl`.

Хотя мы написали лишь часть кода (просто несколько функций, состоящих из одной строки), уже можно кое-что проверить.

```
// Предположим, что файл Functor.h включен
// в реализацию класса Functor
#include "Functor.h"
#include <iostream>
// Для небольших программ на C++ можно применять
// директиву using
using namespace std;

// Определяем тестируемый функтор
struct TestFunctor
{
   void operator()(int i, double d)
   {
      cout << "TestFunctor::operator()(" << i
         << ", " << d << ") called. \n";
   }
};

int main()
{
   TestFunctor f;
   Functor<void, TYPELIST_2(int, double)> cmd(f);
   cmd(4, 4.5);
}
```

Эта маленькая программа выводит на печать следующую строку.

```
TestFunctor::operator()(4, 4.5) called.
```

Следовательно, мы достигли своей цели. Теперь можно идти дальше.

5.7. Один пишем, два в уме

Читая предыдущий раздел, не задавались ли вы вопросом: "Почему мы не начали с реализации простых указателей на обычные функции?". Зачем мы перескочили сразу к

функторам, шаблонам и т.п.? Ответ прост — в настоящее время поддержка указателей на обычные функции *уже реализована*. Изменим немного нашу тестовую программу.

```
#include "Functor.h"
#include <iostream>
using namespace std;

// Определяем тестируемую функцию
void TestFunction (int i, double d)
{
     cout << "TestFunction(" << i
         << ", " << d << ") called. \n";
}

int main()
{
   Functor<void, TYPELIST_2(int, double)> cmd(TestFunction);
   // Выведет на печать строку
   // "TestFunction(4, 4.5) called."
   cmd(4, 4.5);
}
```

Объяснение этого приятного сюрприза кроется в способе вывода шаблонных параметров. Когда компилятор обнаруживает класс `Functor`, созданный из функции `TestFunction`, он вынужден проверить шаблонный конструктор. Затем компилятор конкретизирует шаблонный конструктор шаблонным аргументом `void (&)(int, double)`, представляющим собой тип значения, возвращаемого функцией `TestFunction`. Конструктор конкретизирует класс `FunctorHandler<Functor<...>, void (&)(int, double)>`. Следовательно, переменная `fun_` в классе `FunctorHandler` также имеет тип `void (&)(int, double)`. При вызове оператора `FunctorHandler<...>::operator()` он передается функции `fun_()`. Это вполне допустимая синтаксическая конструкция, означающая вызов функции с помощью указателя. Итак, класс `FunctorHandler` поддерживает указатели на внешние функции по двум причинам: благодаря синтаксической схожести указателей на функции и функторов и особенностям механизма вывода типов в языке C++.

Однако есть одна проблема. (У всех есть свои недостатки, не так ли?) При перегрузке функции `TestFunction` (или любой другой функции, которая передается классу `Functor<...>`) возникает неопределенность. Ее причина заключается в том, что при перегрузке функции `TestFunction` тип символа `TestFunction` становится неопределенным. Чтобы проиллюстрировать этот факт, поместим перегруженную версию функции `TestFunction` прямо перед функцией `main`.

```
// Объявление перегруженной функции TestFuction
// (определение не обязательно)
void TestFunction(int);
```

Неожиданно компилятор сообщает, что не может распознать, какую из перегруженных версий функции `TestFunction` следует использовать. Поскольку есть две функции с именем `TestFunction`, одного имени для идентификации функции недостаточно.

В сущности, при перегрузке есть два способа идентифицировать конкретную функцию: с помощью инициализации (или присваивания) или с помощью приведения типов. Проиллюстрируем оба метода.

```
// Как и выше, функция TestFunction перегружена
int main()
{
   // Для удобства используется оператор typedef
```

```
    typedef void (*TpFun)(int, double);
    // Способ 1: инициализация
    TpFun pF = TestFunction;
    Functor<void, TYPELIST_2(int, double)>
    cmd1(4, 4.5);
    // Способ 2: приведение типов
    Functor<void, int, double)> cmd2(
       static_cast<TpFun>(TestFunction)); // Все правильно!
    cmd2(4, 4.5);
}
```

Оба способа инициализации позволяют компилятору распознать требуемую версию функции TestFuction, которая должна получать параметры типа int и double, а возвращать — значение типа void.

5.8. Преобразование типов аргументов и возвращаемого значения

В идеале преобразование типов для функторов должно выполняться так же, как и при вызовах обычных функций. Тогда можно было бы написать примерно такой код.

```
#include <string>
#include <iostream>
#include "Functor.h"
using namespace std;

// Аргументы игнорируются — в данном примере
// они не представляют интереса
const char* TestFunction(double, double)
{
   static const char buffer[] = "Hello, world!";
   // Можно вернуть указатель на статический буфер
   return buffer;
}

int main()
{
   Functor<string TYPELIST_2(int, int)> cmd(TestFunction);
   // Должен выводить на печать строку "world!"
   cout << cmd(10, 10).substr(7);
}
```

Несмотря на то что сигнатура фактической функции TestFunction немного отличается (она получает два параметра типа double и возвращает константу типа char*), ей можно поставить в соответствие функтор Functor<string, TYPELIST_2(int, int)>. Предполагается, что это возможно, поскольку тип int можно неявно преобразовать в тип double, а значение типа const char* — в тип string. Если класс Functor не поддерживает те же преобразования, что и язык C++, то такие жесткие ограничения следует считать неоправданными.

Для того чтобы удовлетворить новые требования, не нужно писать новый код. Приведенный выше пример компилируется и выполняется без проблем. Почему? Как и прежде, ответ следует искать в определении класса FunctorHandler.

Посмотрим, что произойдет после конкретизации шаблона в приведенном выше примере. Функция

```
string Functor<...>::operator()(int i, int j)
```

передается виртуальной функции

```
string FunctorHandler<...>::operator()(int i, int j)
```

Реализация виртуальной функции выполняет вызов

```
return fun_(i, j);
```

Здесь функция fun_ имеет тип const char*(*)(double, double) и вычисляет значение функции TestFunction.

Когда компилятор обнаружит вызов функции fun_, он нормально его скомпилирует, как если бы вы написали этот вызов собственноручно. Слово "нормально" означает, что правила приведения типов применяются как обычно. Затем компилятор генерирует код, преобразовывающий переменные i и j в тип double, а результат приводится к типу std::string.

Универсальность и гибкость класса FunctorHandler иллюстрирует мощь генерации кода. Синтаксическая замена аргументов шаблона соответствующими параметрами позволяет оперировать с программой на уровне исходного текста. В отличие от этого, мощь объектно-ориентированного программирования проявляется в позднем (после компиляции) связывании имен с их значениями. Таким образом, объектно-ориентированное программирование способствует повторному использованию кода в форме бинарных компонентов, в то время как обобщенное программирование поощряет повторное использование исходного кода. Поскольку исходный код по определению более информативен и находится на более высоком уровне, чем бинарный, обобщенное программирование позволяет создавать более сложные конструкции. Однако это происходит за счет замедления выполнения программы. Со стандартной библиотекой шаблонов невозможно работать так же, как с технологией CORBA, и наоборот. Эти две технологии дополняют друг друга.

Теперь мы можем работать с функторами всех видов и обычными функциями, используя один и тот же базовый код. В качестве вознаграждения мы получаем возможность неявно преобразовывать типы аргументов и возвращаемого значения.

5.9. Указатели на функции-члены

Указатели на функции-члены не часто встречаются в программистской практике, но иногда они оказываются очень полезными. Они аналогичны указателям на обычные функции, но при вызове функции-члена нужно передавать объект (помимо аргументов). Синтаксис и семантика указателей на функции-члены описывается следующим примером.

```
#include <iostream>
using namespace std;

class Parrot
{
   public:
     void Eat()
     {
        cout << "Ням, ням, ням ...\n";
     }
     void Speak()
     {
        cout << "Пиастры! Пиастры!\n";
     }
};
```

```
int main()
{
   // Определяем тип: указатель на функцию-член
   // класса Parrot, не имеющую параметров и
   // возвращающую значение типа void
   typedef void (Parrot::* TpMemFun)();

   // Создаем объект указанного типа
   // и инициализируем его адресом функции Parrot::Eat
   TpMemFun pActivity = &Parrot::eat;
   // Создаем объект класса Parrot
   Parrot geronimo;
   // ... и указатель на него
   Parrot* pGeronimo = &geronimo;

   // С помощью объекта вызываем функцию-член,
   // адрес которой хранится в указателе pActivity.
   // Обратите внимание на оператор .*
   (geronimo.*pActivity)();

   // Делаем то же самое, но с помощью указателя
   (pGeronimo->*pActivity)();
   // Изменяем действие
   pActivity = &Parrot::Speak;

   // Проснись, Джеронимо!
   (geronimo.*pActivity)();
}
```

Приглядевшись внимательнее к указателям на функции-члены и двум связанным с ними операторам — `.*` и `->*`, можно заметить странные особенности. В языке C++ не существует типа для результата работы функций `geronimo.*pActivity` и `geronimo->*pActivity`. С обеими бинарными операциями все в порядке. Они возвращают нечто, к чему можно непосредственно применять оператор вызова функции, но это "нечто" не имеет типа.[5] Результат выполнения операторов `.*` и `->*` невозможно хранить ни в каком виде, хотя он представляет собой сущность, обеспечивающую связь между объектом и указателем на функцию-член. Эта связь выглядит очень непрочной. Она возникает из небытия сразу после вызова операторов `.*` или `->*`, существует довольно долго для того, чтобы к ней можно было применить оператор `()` и возвращается в обратно небытие. Больше с ней ничего сделать нельзя.

В языке C++, в котором каждый объект имеет тип, результат работы операторов `.*` или `->*` является единственным исключением. Понять его еще сложнее, чем неоднозначность указателей на функции, вызванную перегрузкой, рассмотренную в предыдущем разделе. Там мы имели слишком много типов, но не могли сделать однозначный выбор; здесь у нас вообще нет ни одного типа. По этой причине указатели на функции-члены и два связанных с ними оператора представляют собой удивительно непродуманную концепцию в языке C++. Кстати, ссылок на функции-члены не бывает (хотя можно создать ссылки на обычные функции).

В некоторых компиляторах языка C++ вводится новый тип, что позволяет хранить результат выполнения оператора `.*` с помощью следующей синтаксической конструкции.

---

[5] В описании стандарта языка C++ сказано: "Если результатом выполнения операторов `.*` и `->` является функция, этот результат может быть использован лишь в качестве операнда оператора вызова функции `()`".

```
// _closure — расширение языка,
// определенное в некоторых компиляторах
void (_closure:: *geronimosWork)() =
   geronimo.*pActivity;
// Вызываем нужную функцию
geronimosWork();
```

Тип значения, возвращаемого функцией geronimoWork, не содержит никакой информации о классе Parrot. Это значит, что в дальнейшем функцию geronimoWork можно связать с другим классом. Все, что для этого нужно — тип возвращаемого значения и аргументы указателя на функцию-член. Фактически это расширение языка представляет собой некую разновидность класса Functor, но его применение ограничено объектами и функциями-членами (на обычные функции и функторы это расширение не распространяется).

Перейдем к реализации связывания указателей с функциями-членами класса Functor. Опыт работы с функторами и функциями подсказывает, что было бы хорошо остаться на высоком уровне абстракции и до поры до времени пренебречь спецификой. В реализации класса MemFunHandler тип объекта (в предыдушем примере им был класс Parrot) является шаблонным параметром. Более того, указатель на функцию-член мы также сделаем шаблонным параметром. Это позволит свободно выполнять автоматическое преобразование типов в реализации класса FunctorHandler, которая приводится ниже. В этой реализации воплощены описанные выше идеи, а также некоторые идеи, уже использованные при разработке класса Functor.

```
template <class ParentFunctor, typename PointerToObj)
   typename PointerToMemFn>
class MemHandler
   : public FunctorImpl
      <
         typename ParentFunctor::ResultType,
         typename ParentFunctor::ParmList
      >
{
public:
   typedef typename ParentFunctor::ResultType ResultType;
   MemFunHandler(const PointerToObj&, PointerToMemFn pMemFun)
   : pObj_(pObj), pMemFn_(pMemFn) {}

MemFunHandler* Clone() const
{ return new MemFunHandler(*this); }

ResultType operator()()
{
   return ((*pObj_).*pMemFn_)();
}

ResultType operator()(typename ParentFunctor::Parm1 p1)
{
   return ((*pObj_).*pMemFn_)(p1);
}

ResultType operator()(typename ParentFunctor::Parm1 p1,
   typename ParentFunctor::Parm2 p2)
{
   return ((*pObj_).*pMemFn_)(p1, p2);
}
```

```
private:
   PointerToObj pObj_;
   PointerToMemFn pMemFn_;
};
```

Почему параметром шаблонного класса MemFunHandler является тип указателя (PointerToObj), а не тип самого объекта? Более естественно выглядела бы такая реализация.

```
template <class ParentFunctor, typename Obj)
   typename PointerToMemFn>
class MemHandler
   : public FunctorImpl
      <
         typename ParentFunctor::ResultType,
         typename ParentFunctor::ParmList
      >
{
private:
   Obj* pObj_;
   PointerToMemFn pMemFn_;
public:
   MemFunHandler(Obj* pObj, PointerToMemFn pMemFun)
   : pObj_(pObj), pMemFn_(pMemFn) {}
   ...
```

Понять такой код было бы проще. Однако первая реализация является более обобщенной. Во вторую реализацию встроен тип "указатель на объект", причем этот указатель представляет собой просто адрес объекта класса Obj и больше ничего. Может ли это создать проблемы?

Конечно, если понадобится применить интеллектуальные указатели на объекты класса MemFunHandler. Убедились? Первая реализация поддерживает интеллектуальные указатели, а вторая — нет. Первая реализация может хранить любой тип, *действующий* как указатель на объект, а вторая хранит и использует только простые указатели. Более того, вторая версия не работает с указателями на константы. Вот как проявляются негативные черты жестко встроенных типов.

Попробуем протестировать только что созданную реализацию на примере класса Parrot.

```
#include "Functor.h"
#include <iostream>
using namespace std;

class Parrot
{
public:
 void Eat()
     {
        cout << "Ням, ням, ням ...\n";
     }
     void Speak()
     {
        cout << "Пиастры! Пиастры!\n";
     }
};
```

```
int main()
{
   Parrot geronimo;
   // Определяем два функтора
   Functor<>
      cmd1(&geronimo, &Parrot::eat),
      cmd2(&geronimo, &Parrot::Speak);
   // Вызываем каждый из них
   cmd1();
   cmd2();
}
```

Поскольку класс `MemFunHandler` должен быть как можно более общим, автоматическое преобразование типов выполняется совершенно свободно — абсолютно так же, как и в классе `FunctorHandler`.

5.10. Связывание

На этом мы могли бы остановиться. Теперь у нас все есть — класс `Functor` поддерживает все вызываемые сущности языка C++, определенные выше, причем делает это весьма успешно. Однако, как мог убедиться Пигмалион, начиная работу, невозможно предсказать, каков будет ее результат.

Поскольку класс `Functor` уже готов, возникают новые идеи. Например, хотелось бы иметь возможность конвертировать тип `Functor` в другой тип. Одним из таких преобразований является *связывание* (binding). Пусть задан класс `Functor`, получающий целые числа. Мы хотим связать одно из этих целых чисел с некоторым фиксированным значением, а второе оставить переменным. Такое связывание порождает новый класс `Functor`, получающий только одно целое число, поскольку второе зафиксировано и, следовательно, известно. Описанное выше связывание иллюстрируется следующим примером.

```
void f()
{
   // Определяем функтор с двумя аргументами
   Functor<void, TYPELIST_2(int, int)> cmd1(something);
   // Связываем первый аргумент со значением 10
   Functor<void, int> cmd2(BindFirst(cmd1, 10));
   // Эквивалентно cmd1(10, 20)
   cmd2(20);
   // Затем связываем первый (и только первый) аргумент
   // функтора cmd2 со значением 30
   Functor<void> cmd3(BindFirst(cmd2, 30));
   // Эквивалентно cmd1(10, 30)
   cmd3();
}
```

Связывание — мощное оружие. Оно позволяет хранить не только вызываемые сущности, но и часть (или все) их аргументы. Это значительно расширяет возможности функторов, поскольку теперь можно упаковывать функции и аргументы, не прибегая к генерации дополнительного кода для их связывания.

Например, представьте себе реализацию операции повтора (redo) в текстовом редакторе. Когда пользователь набирает букву "а", выполняется функция `Document::InsertChar('a')`. Добавим готовый функтор, содержащий указатель на класс `Document`, функцию-член `InsertChar` и фактический символ. Если пользователь выполняет в меню пункт Redo, достаточно лишь активизировать функтор — и все. Более подробно операции отката и повтора обсуждаются в разделе 5.14.

Связывание является мощным средством еще и по другой причине. Представьте, что вы применяете класс `Functor` для вычислений, а его аргументы — это *окружение*, необходимое для их выполнения. До сих пор класс `Functor` задерживал вычисления, сохраняя указатели на функции и указатели на методы. Однако в классе `Functor` хранятся лишь вычисления, но нет информации об их окружении. Связывание позволяет функтору хранить часть окружения вместе с вычислениями и сократить потребность в пересылке переменных окружения во время вызова.

Перед тем как перейти к реализации, подытожим наши требования. В конкретизации класса `Functor<R, TList>` мы хотим связать первый аргумент (`TList::Head`) с фиксированным значением. Следовательно, типом возвращаемого значения является класс `Functor<R, TList::Tail>`.

Реализовать класс `BinderFirst` очень легко. Нужно лишь учесть, что здесь используются две конкретизации класса `Functor`: входная и результирующая. Входной тип `Functor` передается в качестве параметра `ParentFunctor`, а результирующий тип `Functor` вычисляется.

```
template <class Incoming>
class BinderFirst
   : public FunctorImpl<typename Incoming::ResultType,
       typename Incoming::Arguments::Tail>
{
   typedef Functor<typename Incoming::ResultType,
      Incoming::Arguments::Tail> Outgoing;
   typedef typename Incoming::Parm1 Bound;
   typedef typename Incoming::ResultType ResultType;

public:
   BinderFirst(const Incoming& fun, Bound bound)
   : fun_(fun), bound_(bound)
   {
   }

   BinderFirst* Clone() const
   { return new BinderFirst(*this); }

   ResultType operator()()
   {
      return fun_(bound_);
   }

   ResultType operator()(typename Outgoing::Parm1 p1)
   {
      return fun_(bound_, p1);
   }

   ResultType operator()(typename Outgoing::Parm1 p1,
      typename Outgoing::Parm2 p2)
   {
      return fun_(bound_, p1, p2);
   }

   private:
      Incoming fun_;
      Bound bound_;
   };
```

Шаблонный класс BinderFirst работает в сочетании с шаблонной функцией BindFirst. Достоинством шаблонной функции BindFirst является то, что она автоматически выводит свои шаблонные параметры из типов получаемых ею фактических параметров.

```
// Определение класса BinderFirstTraits
// находится в файле Functor.h
template <class Fctor>
typename Private::BinderFirstTraits<Fctor>::BoundFunctorType
BindFirst(
   const Fctor& fun,
   typename Fctor::Parm1 bound)
{
   typedef typename
   private::BinderFirstTraits<Fctor>::BoundFunctorType
      Outgoing;
   return Outgoing(std::auto_ptr<typename Outgoing::Impl>(
     new BinderFirst<Fctor>(fun, bound)));
}
```

Связывание прекрасно сочетается с автоматическим преобразованием типов, наделяя класс Functor невероятной гибкостью. Это иллюстрирует следующий пример.

```
const char* Fun(int i, int j)
{
   cout << Fun(" << i << ", " << j << ") called\n";
}

int main()
{
   Functor<const char*, TYPELIST_2(char, int)> f1(Fun);
   Functor<std::string, TYPELIST_1(double)> f2(
     BindFirst(f1, 10));
   // Выводит на печать строку "Fun(10, 15) called"
   f2(15);
}
```

5.11. Сцепление

В книге Gamma et al. (1995) приведен пример класса MacroCommand, в котором хранится линейный набор (список или вектор) объектов класса Command. При выполнении этого класса последовательно выполняются все хранящиеся в нем команды.

Это свойство может оказаться очень полезным. Например, вернемся к примеру, связанному с откатом и повторным выполнением операции undo/redo. Каждая операция "do" должна сопровождаться несколькими операциями отката "undo". Например, вставка символа может автоматически приводить к прокрутке текстового окна (в некоторых текстовых редакторах эта операция применяется для улучшения внешнего вида текста). Отменяя эту операцию, вы возвращаете окно в прежнее положение. (В большинстве текстовых редакторов эта операция реализована неправильно. Досадно!) Чтобы выполнить обратную прокрутку (unscroll), нужно хранить несколько объектов класса Command в одном объекте класса Functor и выполнять их как одно целое. Функция-член Document::InsertChar заталкивает объект класса MacroCommand в стек отката. В состав класса MacroCommand могут входить функции-члены Document::DeleteChar и Window::Scroll. Последнюю функцию можно связать с аргументом, запоминающим старое состояние. (И вновь связывание оказывается очень удобным инструментом.)

В библиотеке Loki определен класс FunctorChain и вспомогательная функция Chain.

```
template <class Fun1, class Fun2>
Fun2 Chain(
   const Fun1& fun1,
   const Fun2& fun2);
```

Реализация класса FunctorChain тривиальна — он хранит два функтора, а оператор FunctorChain::operator() вызывает их один за другим.

Функция Chain завершает наше описание поддержки макрокоманд. Одно из достоинств такой реализации заключается в том, что список команд неограничен. Ни шаблонная функция BindFirst, ни функция Chain не требуют вносить никаких изменений в класс Functor. Это свидетельствует о том, что вы сами можете разработать аналогичные функциональные возможности.

5.12. Первая практическая проблема: стоимость функций пересылки

В принципе разработка шаблонного класса Functor завершена. Теперь мы сосредоточимся на вопросах его оптимизации, стремясь, чтобы он работал как можно эффективнее.[6]

Рассмотрим в классе Functor одну из перегрузок оператора (), пересылающую вызов интеллектуальному указателю.

```
// Внутри класса Functor<R, TList>
R operator()(Parm1 p1, Parm2 p2)
{
   return (*spImpl_)(p1, p2);
}
```

При каждом вызове оператора () создается ненужная копия каждого аргумента. Если классы Parm1 и Parm2 достаточно велики, это снизит производительность работы программы.

Довольно странно, но даже если оператор () класса Functor сделать подставляемым (inline), компилятор будет по-прежнему тиражировать лишние копии аргументов. В задаче 46 книги Саттера (Sutter, 2000) описывается современная модификация языка, сделанная непосредственно перед его стандартизацией. Функциям пересылки было *запрещено создавать копии* (eliding for copy construction). Компилятору позволено создавать копии лишь для возвращаемого значения, поскольку возврат значений нельзя оптимизировать вручную.

Помогут ли ссылки решить эту проблему? Используем следующий код.

```
// Внутри класса Functor<R, TList>
R operator()(Parm1& p1, Parm2& p2)
{
    return (*spImpl_)(p1, p2);
}
```

---

[6] Обычно преждевременная оптимизация нежелательна. Одна из причин заключается в том, что программисты не могут правильно определить, какую часть программы следует оптимизировать, а какую — нет (что еще важнее). Однако разработчики библиотек находятся в другой ситуации. Они не знают, будут или нет использованы их библиотеки в критически важных частях какого-то приложения, поэтому стремятся максимально оптимизировать библиотеку.

Все это выглядит прекрасно и может на самом деле работать, пока вы не попробуете выполнить такой код.

```
void testFunction(std::string&, int);
Functor<void, TYPELIST_2(std::string&, int)>
   cmd(testFunction);
...
string s;
cmd(s, 5); // Ошибка!
```

Компилятор споткнется на последней строке, выдав сообщение: "Ссылки на ссылки не допускаются!". (На самом деле сообщение может быть более таинственным.) Фактически такая конкретизация может преобразовать класс Parm1 в ссылку на объект класса std::string. Следовательно, объект p1 может стать ссылкой на ссылку на объект класса std::string, а это не допускается.[7]

К счастью, решение этой проблемы существует. В главе 2 описан шаблонный класс TypeTraits<T>, определяющий группу типов, связанных с типом T. К ним относятся неконстантные типы (если тип T является константным), указатели (если тип T является указателем) и многие другие. Тип, который можно безопасно и эффективно передавать как параметр функции, называется ParameterType. В приведенной ниже таблице показана связь между типом, передаваемым классу TypeTraits, и внутренним определением типа ParameterType.

T	TypeTraits<T>::ParameterType
U	Тип U, если U — элементарный тип; в противном случае — const U &
const U	Тип U, если U — элементарный тип; в противном случае — const U &
U &	U &
const U &	const U &

Замена аргументов функции пересылки типами, указанными в правом столбце, всегда корректна, причем она не вызывает никаких дополнительных затрат, связанных с копированием.

```
// Внутри класса Functor<R, TList>
R operator()(
   typename TypeTraits<Parm1>::ParameterType p1;
   typename TypeTraits<Parm2>::ParameterType p2;
{
   return (*spImpl_)(p1, p2);
}
```

Еще приятнее то, что эти ссылки прекрасно работают в сочетании с подставляемыми функциями. Оптимизатор легче генерирует оптимальный код, поскольку для этого ему достаточно установить ссылки.

[7] Эта проблема также возникает при работе со стандартными механизмами связывания. Бьярн Страуструп представил Комитету по Стандартизации отчет об этом дефекте. Он предложил исправить этот недостаток, разрешив ссылаться на ссылки и обрабатывать результирующие ссылки как простые. В момент написания книги этот отчет был доступен на Web-странице http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/cwg_active.html#106.

5.13. Вторая практическая проблема: распределение динамической памяти

Оценим затраты, связанные с созданием и копированием функторов. Мы должны реализовать правильную семантику значений, но это связано с проблемой распределения динамической памяти. Каждый объект класса Functor содержит интеллектуальный указатель на объект, созданный с помощью оператора new. Для создания копии объекта класса Functor функция-член FunctorImpl::Clone выполняет глубокое копирование.

Это особенно неприятно, если размер объектов очень важен. В большинстве случаев класс Functor используется с указателями на функции и парами, состоящими из указателей на объекты и указателей на функции-члены. В типичных 32-разрядных системах эти объекты занимают от 4 до 20 байт соответственно (4 байт для указателя на объект и 16 байт для указателя на функцию-член[8]). При использовании связывания размер конкретного функтора увеличивается примерно на величину связываемого аргумента (вследствие выравнивания блоков памяти его размер может еще ненамного вырасти).

В главе 4 описаны эффективные механизмы распределения памяти для небольших объектов. Класс FunctorImpl и его наследники являются превосходными кандидатами на применение этих механизмов распределения памяти. Напомним, что один из способов применения механизма распределения памяти для небольших объектов заключается в создании класса, производного от шаблонного класса SmallObject.

Использовать этот класс очень легко. Однако нам нужно добавить в класс Functor шаблонный параметр, отображающий потоковую модель в механизме распределения памяти. Это не сложно, поскольку большую часть времени используется аргумент, заданный по умолчанию. В приведенном ниже коде изменения выделены полужирным шрифтом.

```
template
<
   class R,
   class TL,
   template <class T>
      class ThreadingModel = DEFAULT_THREADING
>
class FunctorImpl : public SmallObject<ThreadingModel>
{
public:
   ... как и раньше ...
};
```

Все это наделяет класс FunctorImpl функциональными возможностями механизма распределения памяти.

Аналогично в сам класс Functor добавляется третий шаблонный параметр.

```
template
<
   class R,
   class TL,
   template <class T>
      class ThreadingModel = DEFAULT_THREADING
```

[8] Естественно было бы ожидать, что указатель на функцию-член занимает 4 байт, как и указатель на обычную функцию. Однако указатели на методы фактически представляют собой размеченные объединения (tagged union). Они применяют множественное виртуальное наследование и виртуальные/невиртуальные функции.

```
>
class Functor
{
    ...
private:
    // Передаем параметр ThreadingModel классу FunctorImpl
    std::auto_ptr<FunctorImpl<R, TL, ThreadingModel> pImpl_;
};
```

Для того чтобы использовать класс Functor со стандартной потоковой моделью, третий шаблонный аргумент не нужен. Только если в приложении понадобятся функторы, поддерживающие несколько потоковых моделей, нужно явно указать параметр ThreadingModel.

5.14. Реализация операций Undo и Redo с помощью класса Functor

В книге Gamma et al (1995) рекомендуется реализовывать операцию undo с помощью дополнительной функции-члена Unexecute класса Command. Проблема заключается в том, что эту функцию невозможно выразить в обобщенном виде, поскольку отношение между некоей операцией и ее отменой (undo) нельзя предсказать. Решить эту задачу можно, если создать отдельный класс Command для каждой операции, выполняемой в приложении. Однако подход, основанный на применении класса Functor, ориентирован на использование одного класса, связываемого с разными объектами и вызовами функций-членов.

Статья Ала Стивенса (Al Stevens) в журнале *Dr. Dobbs Journal* (Stevens, 1998) может оказать нам огромную помощь в изучении обобщенных реализаций операций undo и redo. Стивенс создал обобщенную библиотеку операций undo/redo, которую следует тщательно изучить, независимо от того, будете вы применять класс Functor или нет.

Это все, что нам следует знать о структурах данных. Основная идея операций undo и redo заключается в использовании стека отката (undo stack) и стека повтора операций (redo stack). Если пользователь выполнил некое действие, например, набрал букву на клавиатуре, в стек отката заталкивается *новый* функтор. Это означает, что функция-член Document::InsertChar должна затолкнуть в стек отката действие, обратное по отношению к операции вставки символа (например, функцию-член Document::DeleteChar). Основная нагрузка переносится на функцию-член, которая действительно выполняет операцию отката, а в классе Functor информация о том, как именно следует выполнять эту операцию, не хранится.

При необходимости можно затолкнуть в стек повторения операции функтор, состоящий из класса Document и указателя на функцию Document::InsertChar, связанных с фактическим символьным типом. Некоторые текстовые редакторы позволяют "повторный набор" (retyping). После того как вы что-то набрали на клавиатуре и выбрали в меню пункт Redo, повторяется блок введенных символов. Для этого прекрасно подходит связывание, реализованное в классе Functor, позволяющее хранить вызов функции-члена Document::InsertChar для заданного символа, инкапсулируя такие вызовы в одном функторе. Кроме того, повторить нужно не только символ, набранный последним (это было бы не слишком впечатляющим достижением), но и всю последовательность символов, набранных после выполнения последней опера-

ции, не связанной с набором. Здесь в действие вступает сцепление, реализованное в классе Functor. По мере того как пользователь вводит все новые и новые символы, создаются все новые и новые функторы. Таким образом, возникает возможность выполнить последовательность нажатий клавиш как одну операцию.

Функция-член Document::InsertChar, по существу, заталкивает функтор в стек отката. Когда пользователь выбирает пункт меню Undo, этот функтор должен быть выполнен и затолкнут в стек повтора операций.

Как видим, связывание аргументов и сцепление позволяют нам работать с функторами единообразно: вид вызова не имеет значения, поскольку он скрыт внутри функтора. Это значительно облегчает задачу реализации операций отката и повтора операций.

5.15. Резюме

Использовать хорошие библиотеки на языке C++ намного легче, чем создавать. С другой стороны, разрабатывать библиотеки очень интересно. Оглядываясь на детали описанных выше реализаций, можно извлечь несколько уроков, касающихся создания обобщенных программ.

- Создавайте гибкие шаблонные типы. Стремитесь на все смотреть с максимально общей точки зрения. Классы FunctorHandler и MemFunHandler приобрели много преимуществ, благодаря тому, что в них используются шаблоны. Указатели на функции предоставляют большую свободу. По сравнению с их функциональными возможностями получающийся код имеет удивительно небольшой размер. Все эти выгоды достигаются благодаря использованию шаблонов и тому, что компилятору предоставлено право самому выводить типы, когда это возможно.
- Обобщенное программирование способствует созданию семантики первого класса (см. примечание в начале главы. — *Прим. ред.*). Было бы крайне затруднительно оперировать исключительно указателями на класс FunctorImpl. Представьте себе, как в этих условиях реализуются связывание и сцепление.

Изощренные технологии предназначены для достижения большей простоты. На основе всех этих шаблонов, наследования, связывания и управления памятью создана простая, легкая в применении и хорошо продуманная библиотека.

В двух словах, класс Functor задерживает вызов функции, функтора или функции-члена. Он сохраняет вызываемый объект и предоставляет для его вызова оператор (). Ниже приведено краткое описание этого класса.

5.16. Краткое описание класса Functor

Класс Functor является шаблонным и позволяет выражать вызовы функций, имеющих до 15 аргументов. Первым шаблонным параметром является тип возвращаемого значения, вторым — список типов, содержащий типы параметров функции. Третий шаблонный параметр задает потоковую модель, которая используется механизмом распределения памяти, применяемым в классе Functor. Подробная информация о списках классов приведена в главе 3, о потоковой модели — в приложении, о механизме распределения памяти для небольших объектов — в главе 4.

- Объект класса Functor можно инициализировать функцией, функтором, другим объектом класса Functor или указателем на объект и указателем на метод, как показано в приведенном ниже примере.

```
void Function(int)

struct SomeFunctor
{
   void operator()(int);
};

struct SomeClass
{
   void MemberFunction(int);
};

void example()
{
   // Инициализируем класс Functor функцией
   Functor<void, TYPELIST_1(int)> cmd1(Function);
   // Инициализируем класс Functor функтором
   SomeFunctor fn;
   Functor<void, TYPELIST_1(int)> cmd2(fn);
   // Инициализируем класс Functor указателем на объект
   // и указателем на функцию-член
   SomeClass myObject;
   Functor<void, TYPELIST_1(int)> cmd3(&myObject,
      &SomeClass::MemberFunction);
   // Инициализируем класс Functor другим объектом
   // класса Functor (копирование)
   Functor<void, TYPELIST_1(int)> cmd4(cmd3);
}
```

- Объект класса `Functor` можно инициализировать объектом класса `std::auto_ptr< FunctorImpl<R, TList> >`. Это позволяет создавать расширения, определенные пользователем.
- Класс `Functor` поддерживает автоматические преобразования типов аргументов и возвращаемых значений. Например, в предыдущем примере функции-члены `SomeFunctor::operator()` и `SomeClass::MemberFunction` могут получать аргументы типа `double` вместо аргументов типа `int`.
- При перегрузке функций-членов следует создавать дополнительный код для устранения неоднозначности.
- Класс `Functor` полностью поддерживает семантику первого класса: копирование, присваивание и передачу по значению. Класс `Functor` не является ни полиморфным, ни базовым. Если нужно расширить класс `Functor`, в качестве базового следует применять класс `FunctorImpl`.
- Класс `Functor` поддерживает связывание аргументов. Вызов шаблонной функции `BindFirst` связывает первый аргумент с фиксированным значением. В результате возникает параметризованный класс `Functor`, параметрами которого являются остальные аргументы. Рассмотрим пример.

```
void f()
{
   // Определяем класс Functor  с тремя аргументами
   Functor<void, TYPELIST_3(int, int, double)> cmd1(
      someEntity);
   // Связываем первый аргумент с числом 10
   Functor<void, TYPELIST_2(int, double)> cmd2(
      BindFirst(cmd1, 10);
```

```
        // Эквивалентно вызову cmd1(10, 20, 5.6)
        cmd2(20, 5.6)
    }
```

- Используя функцию Chain, в одном объекте класса Functor можно сцепить несколько объектов этого класса.

```
    void f()
    {
        Functor<> cmd1(something);
        Functor<> cmd2(somethingElse);
        // Сцепливаем объекты cmd1 и cmd2 в контейнер
        Functor<> cmd3(Chain(cmd1, cmd2));
        // Эквивалентно cmd1(); cmd2();
        cmd3();
    }
```

- Затраты, связанные с использованием простых объектов класса Functor, состоят из одной переадресации (вызова через указатель). При каждом связывании или сцепливании возникает один дополнительный виртуальный вызов. Параметры копируются только при необходимости преобразования типов.
- Шаблонный класс FunctorImpl использует механизм распределения памяти для небольших объектов, описанный в главе 4.

6

Реализация шаблона Singleton

Шаблон проектирования Singleton (Gamma et al., 1995) представляет собой уникальную комбинацию простого описания и сложной реализации. Об этом свидетельствуют многочисленные книги и журнальные публикации (например, Vlissides, 1996, 1998). Описание шаблона Singleton в книге Gamma et al. (1995) весьма просто: "Шаблон гарантирует, что класс имеет только один экземпляр и обеспечивает глобальный доступ к нему".

Синглтон — это улучшенный вариант глобальной переменной. Новшество заключается в том, что второй объект типа Singleton создать нельзя. Следовательно, синглтоны можно использовать в приложениях при моделировании типов, имеющих только один экземпляр, таких как Keyboard, Display, PrintManager и SystemClock. Создавать вторые экземпляры таких классов совершенно неестественно и, кроме того, опасно.

Глобальный доступ имеет скрытые побочные последствия — с точки зрения клиента объект класса Singleton *владеет самим собой*. Для создания объекта класса Singleton клиент ничего не должен делать. Следовательно, объект класса Singltone сам создает и разрушает себя. Управление продолжительностью жизни синглтона представляет собой головоломную задачу.

В этой главе обсуждаются наиболее важные вопросы, связанные с проектированием и реализацией разных вариантов класса Singleton в языке C++.

- Свойства, которые отличают синглтон от обычной глобальной переменной.
- Основные идиомы языка C++ для поддержки синглтонов.
- Обеспечение уникальности синглтона.
- Уничтожение синглтона и распознавание доступа после уничтожения.
- Усовершенствованное управление продолжительностью жизни объекта класса Singltone.
- Многопоточность.

Мы опишем способы решения перечисленных выше проблем, а затем применим их для реализации обобщенного шаблонного класса SingletonHolder.

"Оптимальной" реализации шаблона проектирования Singleton не существует. В зависимости от конкретной задачи наилучшими оказываются различные реализации, включая машинно-зависимые. Подход, описанный в этой главе, предназначен для разработки семейства реализаций на основе обобщенного скелета класса в духе проектирования на основе стратегий (глава 1). Класс SingletonHand оставляет также возможности для его расширения и настройки.

В конце этой главы мы разработаем шаблонный класс SingletonHolder, способный генерировать много синглтонов разного типа. Класс SingletonHolder предостав-

ляет программисту возможности для детального управления распределением памяти при создании и уничтожении синглтонов при условии, что это не создает опасности для потоков. Кроме того, класс определяет, что делать, если пользователь попытался обратиться к уже разрушенному синглтону. Шаблонный класс SingletonHolder предоставляет программисту функциональные возможности, выходящие за рамки обычных типов, определенных пользователем.

6.1. Статические данные + статические функции != синглтон

На первый взгляд кажется, что для создания класса Singleton нужны лишь статические функции-члены и переменные-члены.

```
class Font { ... };
class PrinterPort { ... };
class PrintJob { ... }

class MyOnlyPrinter
{
   public:
      static void AddPrintJob(PrintJob& newJob)
      {
         if(printQueue_.empty() && printingPort_.available())
         {
            printingPort_.send(newJob.Data());
         }
         else
         {
            printQueue_.push(newJob);
         }
      }
private:
   // Все данные являются статическими
   static std::queue<PrintJob> printQueue_;
   static PrinterPort printingPort_;
   static Font defaultFont_;
}

PrintJob somePrintJob("MyDocument.txt");
MyOnlyPrinter::AddPrintJob(somePrintJob);
```

Однако это решение[1] в некоторых ситуациях имеет много недостатков. Основная проблема заключается в том, что *статические функции не могут быть виртуальными*, что не позволяет изменять поведение объекта при открытии кода класса MyOnlyPrinter.

Кроме того, инициализация и удаление объекта затрудняются. В классе MyOnlyPrinter нет конкретных точек, в которых инициализируются и удаляются данные, хотя эти задачи могут быть нетривиальными — например, переменная-член defaultFont_ может зависеть от скорости обмена данными порта, поведение которого описывается переменной printingPort_.

Таким образом, реализация синглтонов сводится к созданию и управлению уникальным объектом, при этом создавать другие объекты такого типа запрещено.

---

[1] На самом деле этот код иллюстрирует шаблон проектирования Monostate (Ball and Crawford, 1998).

6.2. Основные идиомы языка C++ для поддержки синглтонов

Чаще всего синглтоны создаются с помощью вариаций следующей идиомы.

```
// Заголовочный файл Singleton.h
class Singleton
{
public:
   static Singleton* Instance() // Единственная точка доступа
   {
      if (!pInstance_)
        pInstance_ = new Singleton;
      return pInstance_;
   }
   ... операции ...
private:
   Singleton(); // Предотвращает создание нового
                // объекта класса Singleton
   Singleton(const Singleton&); // Предотвращает создание
                // копии объекта класса Singleton
   static Singleton&* pInstance_; // Единственный экземпляр
};

// Файл реализации Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

Поскольку все конструкторы закрыты, код пользователя не может создать объект класса `Singleton`. Однако класс содержит функции-члены, в частности, функцию `Instance`, которые *позволяют* создавать объекты. Следовательно, требование уникальности объекта класса `Singleton` накладывается на этапе компиляции. В этом и заключается суть реализации шаблона проектирования `Singleton` в языке C++.

Если объект класса `Singleton` никогда не используется (в программе нет вызовов функции `Instance`), он не создается. Для осуществления такой оптимизации в начале функции `Instance` выполняется дополнительная проверка (обычно весьма несложная). Преимущество такого подхода становится значительным, если объект класса `Singleton` требует для своего создания больших затрат и в то же время редко используется.

Есть большое искушение упростить класс, заменив указатель `pInstance_` в предыдущем примере полноценным объектом класса `Singleton`.

```
//Заголовочный файл Singleton.h
class Singleton
{
public:
  static Singleton* Instance() // Единственная точка доступа
  {
     return &instance_;
  }
  int DoSomething();
private:
   static Singleton instance_;
};
// Файл реализации Singleton.cpp
Singleton Singleton::instance_;
```

Это неудачное решение. Несмотря на то что переменная `instance_` является статическим членом класса `Singleton` (как и указатель `pInstance_` в предыдущем при-

мере), между этими вариантами есть существенная разница. Переменная-член `instance_` инициализируется *динамически* (с помощью вызова конструктора класса `Singleton` во время выполнения программы), в то время как переменная `pInstance_` *инициализируется статически* (она относится к типу, не имеющему конструктора, и инициализируется статической константой).

Компилятор выполняет статическую инициализацию до первого оператора компоновки (assembly operator) программы. (Обычно статическая инициализация соответствует файлу, содержащему выполняемую программу, т.е. загрузка — это инициализация.) С другой стороны, в языке C++ не определен порядок динамической инициализации объектов, принадлежащих разным транслируемым модулям (translation units), что приводит к многочисленным недоразумениям. (Прще говоря, *транслируемый модуль* — это компилируемый исходный файл.) Рассмотрим следующий код.

```
// SomeFile.cpp
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething();
```

В зависимости от порядка инициализации, выбранного компилятором для переменных `instance_` и `global`, вызов функции-члена `Singleton::Instance` может возвращать объект, который еще не создан. Это значит, что переменную `instance_` нельзя считать проинициализированной, если ее использует другой внешний объект.

6.3. Обеспечение уникальности синглтонов

Обеспечить уникальность объекта класса `Singleton` можно несколькими способами. Некоторые из них мы уже применяли. Это *закрытые конструктор по умолчанию* и *конструктор копирования*. Их применение блокирует код, приведенный ниже.

```
Singleton sneaky(*Singleton::Instance()); // Ошибка!
// Не позволяет создать копию sneaky объекта класса Singleton,
// возвращаемого функцией-членом Instance
```

Если не определить конструктор копирования, компилятор сам создаст открытый конструктор (Meyers, 1998a). Объявление явного конструктора копирования блокирует автоматическую генерацию, и размещение конструктора в разделе `private` вызовет ошибку на этапе компиляции.

Ситуацию можно немного исправить, если позволить функции-члену `Instance` возвращать ссылку, а не указатель. Проблема, связанная с указателем, возвращаемым функцией-членом `Instance`, заключается в том, что вызывающий код может попытаться удалить его с помощью оператора `delete`. Для того чтобы минимизировать вероятность этого события, безопаснее возвращать ссылку.

```
// Внутри класса Singleton
static Singleton& instance();
```

Другая функция-член, генерируемая компилятором по умолчанию, представляет собой оператор присваивания. Уникальность объекта не связана с оператором присваивания непосредственно, однако одно из ее очевидных последствий заключается в том, что присваивать один объект другому нельзя, поскольку два объекта класса `Singleton` одновременно существовать не могут. Для объекта класса `Singleton` любое присваивание оказывается самоприсваиванием (self-assignment), не имеющим никакого смысла. Следовательно, следует заблокировать оператор присваивания (сделав его закрытым и никогда не реализуя).

Последний уровень защиты создается закрытым деструктором. Эта мера предотвращает случайное удаление указателя на объект класса Singleton.

Таким образом, интерфейс класса Singleton принимает следующий вид.

```
class Singleton
{
   Singleton& Instance();
   ... операции ...
private:
   Singleton();
   Singleton(const Singleton&);
   Singleton& operator=(const Singleton&);
   ~Singleton();
}
```

6.4. Разрушение объектов класса Singleton

Как указывалось выше, объекты класса Singleton создаются по требованию при первом вызове функции-члена Instance, который определяет момент создания, но оставляет открытым вопрос об удалении объекта. Когда следует удалять созданный экземпляр, в книге Gamma et al. (1995) не указано, однако, по свидетельству Джона Влиссидеса (*Pattern Hatching*, 1998), это трудная задача.

Действительно, если объект класса Singleton не удален, то это не означает утечки памяти (memory leak). Подобная проблема возникает, когда в памяти накапливаются данные, ссылки на которые утеряны. Ситуация, связанная с объектом класса Singleton, совершенно иная. Здесь ничего не накапливается, и адрес объекта известен до самого конца работы приложения. Более того, все современные операционные системы сами выполняют полную очистку памяти при завершении программы. (Интересное обсуждение проблемы, что считать, а что не считать утечкой памяти, проведено в книге *Effective C++* (Meyers, 1998a).)

Однако утечка памяти все же *имеет место*, более того, происходит *скрытая утечка ресурсов* (insidious resourse leak). Конструктор класса Singleton может запросить неограниченное количество ресурсов: сетевые соединения, обработку системных мьютексов (OS-wide mutexes) и другие внутрипроцессные средства связи, ссылки на внепроцессные (out-of-process) CORBA- или COM-объекты и т.п.

Единственный способ избежать утечки ресурсов — удалить объект класса Singleton в момент завершения работы приложения. Вопрос заключается в том, как выбрать правильный момент для удаления объекта, чтобы не возникло обращения к уже удаленному объекту.

Проще всего при удалении синглтона положиться на языковые механизмы. Например, приведенный ниже код демонстрирует другой подход к реализации синглтона. Вместо размещения объекта в динамической памяти и применения статического указателя, функция Instance использует *локальную статическую переменную* (local static variable).

```
Singleton& Singleton::Instance()
{
   static Singleton obj;
   return obj;
}
```

Эта простая и элегантная реализация была впервые опубликована Скоттом Мейерсом (Meyers, 1996, Item 26), поэтому мы будем называть такой класс *синглтоном Мейерса*.

Синглтон Мейерса использует некоторые скрытые свойства компилятора. Статические объекты, определенные в теле функции, инициализируются, когда поток управления впервые проходит через ее определение. Не путайте статические переменные, которые инициализируются во время выполнения программы, с элементарными статическими переменными, которые инициализируются статическими константами. Рассмотрим следующий пример.

```
int Fun()
{
   static int x = 100;
   return ++x;
}
```

В данном случае переменная x инициализируется до выполнения первого оператора программы. При первом вызове функции Fun известно лишь, что переменная x давным-давно равна 100. В противном случае, когда переменная инициализируется не константой периода компиляции, или статическая переменная является объектом, имеющим конструктор, локальная статическая переменная инициализируется во время выполнения программы при первом проходе потока управления через определение функции.

Кроме того, компилятор так генерирует код, что после инициализации система поддержки выполнения программ регистрирует переменную как предназначенную к удалению. Этот алгоритм можно выразить с помощью следующего псевдокода. (Переменные, имя которых содержит два подчеркивания, считаются скрытыми, т.е. они генерируются и управляются только компилятором.)

```
Singleton& Singleton::Instance()
{
   // Функции, генерируемые компилятором
   extern void __ConstructSingleton(void* memory);
   extern void __DestroySingleton();
   // Переменные, генерируемые компилятором
   static bool __initialized = false;
   // Буфер, в котором хранится синглтон
   // (предполагается, что буфер выровнен)
   static char __buffer[sizeof(Singleton)];
   if (! __initialized)
   {
      // Первый вызов, создается объект
      // Вызывается функция Singleton::Singleton
      // в буферной памяти __buffer
      __ConstructSingleton(__buffer)
      // Регистрируется удаление
      atexit(__DestroySingleton);
      __initialized = true;
   }
   return *reinterpret_cast<Singleton *>(_buffer);
}
```

Сердцевиной этого кода является вызов стандартной функции atexit, позволяющей автоматически вызывать зарегистрированные ею функции при выходе из программы в порядке LIFO (last in, first out — последним пришел, первым ушел). (По определению удаление объектов в языке C++ выполняется по принципу LIFO: объекты, созданные первыми, разрушаются последними. Разумеется, объекты, явно управляемые с помощью операторов new и delete, этому правилу не подчиняются.) Сигнатура функции atexit имеет следующий вид.

```
// Получает указатель на функцию.
// Возвращает 0, если работа выполнена успешно,
// или ненулевое значение, если возникла ошибка
int atexit(void(*pFun)());
```

Компилятор генерирует функцию __DestroySingleton, разрушающую объект класса Singleton, хранящийся в буфере __buffer, и передает ее адрес функции atexit.

Как работает функция atexit? При каждом вызове этой функции ее параметр заталкивается в закрытый стек, предусмотренный библиотекой поддержки выполнения программ на языке C (C runtime library). При выходе из приложения система поддержки выполнения программ вызывает функции, зарегистрированные функцией atexit.

Вскоре мы увидим, насколько важна функция atexit, а также (иногда, к сожалению), насколько она тесно связана с реализацией шаблона проектирования Singleton на языке C++. Нравится вам это или нет, она останется в центре внимания до конца главы. Независимо от того, каким образом реализуется удаление объекта класса Singleton, для его выполнения необходима функция atexit.

Синглтоны Мейерса предоставляют простейшие средства для удаления объектов класса Singleton при выходе из программы. В большинстве случаев они работают безупречно. Мы хорошо их изучим, внесем в них некоторые усовершенствования и создадим альтернативные реализации для специальных ситуаций.

6.5. Проблема висячей ссылки

Чтобы конкретизировать наше обсуждение, рассмотрим пример, который мы будем в дальнейшем использовать для иллюстрации разных реализаций. Как и шаблон проектирования Singleton, этот пример легко сформулировать и понять, но трудно реализовать.

Допустим, что в нашем приложении используются три синглтона: Keyboard, Display и Log. У первых двух моделей есть физические прототипы. Синглтон Log предназначен для генерации сообщений об ошибках. Он может воплощаться в виде текстового файла либо вспомогательной консоли.

Будем предполагать, что конструкция синглтона Log вынуждает затрачивать определенный объем дополнительных ресурсов, поэтому его лучше всего конкретизировать только в случае возникновения ошибки. Таким образом, если в программе нет ошибок, она вообще не создает объект класса Log.

Программа пересылает объекту класса Log все ошибки, возникающие при конкретизации классов Keyboard и Display, а также при уничтожении их объектов.

Если мы попытаемся реализовать эти классы в виде синглтонов Мейерса, программа будет работать неправильно. Например, допустим, что после успешного создания объекта класса Keyboard инициализация объекта класса Display потерпела неудачу. Конструктор объекта класса Display создает объект класса Log, в нем регистрируется ошибка, и похоже, что на этом приложение должно завершить свою работу. При выходе из программы вступают в действие правила языка: система поддержки выполнения программ разрушает локальные статические объекты в порядке, обратном очередности их создания. Следовательно, объект класса Log будет разрушен *до* объекта класса Keyboard. Если по некоторой причине объект класса Keyboard не будет успешно уничтожен, а попытается передать объекту класса Log сообщение о возникшей ошибке, функция-член Log::Instance невольно вернет ссылку на "оболочку" разрушенного объекта класса Log. Таким образом, поведение программы станет непредсказуемым. В этом и заключается проблема висячей ссылки (dead-reference problem).

Порядок создания и разрушения объектов классов Keyboard, Log и Display заранее не определен. Необходимо, чтобы классы Keyboard и Display подчинялись правилам языка C++ (последним создан — первым уничтожен), а класс Log должен быть исключением из этих правил. Независимо от того, когда он был создан, объект класса Log всегда должен уничтожаться после объектов классов Keyboard и Display, чтобы он мог получать сообщения об ошибках, возникающих при их разрушении.

Если приложение содержит несколько взаимодействующих синглтонов, автоматически проконтролировать продолжительность их жизни невозможно. Нормальный синглтон должен по крайней мере *распознавать* висячие ссылки. Этого можно достичь, отслеживая процесс разрушения объектов с помощью статической булевской переменной destroyed_. Начальное значение этой переменной равно false. Деструктор класса Singleton присваивает ей значение true.

Перед тем как перейти к реализации, подведем итоги. Кроме создания объекта класса Singleton и возврата ссылки на него, функция-член Singleton::Instance должна распознавать висячую ссылку. Следуя принципу "одна функция — одна задача", реализуем три разные функции-члена: Create, фактически создающую объект класса Singleton, OnDeadReference, выполняющую обработку ошибок, и хорошо известную функцию Instance, предоставляющую доступ к объекту класса Singleton. Из всех этих функций только Instance является открытой.

Реализуем класс Singleton, распознающий висячую ссылку. Во-первых, добавим в класс Singleton статическую булевскую переменную-член destroyed_. Она предназначена для индикации висячей ссылки. Затем изменим деструктор класса Singleton, чтобы он присваивал переменной pInstance_ значение 0, а переменной destroyed_ — значение true. Новый класс и функция OnDeadReference имеют следующий вид.

```
// Singleton.h
class Singleton
{
public:
   Singleton& Instance()
   {
      if (!pInstance_)
      {
         // Проверка висячей ссылки
         if (destroyed_)
         {
            OnDeadReference();
         }
         else
         {
            // Первый вызов — инициализация
            Create();
         }
      }
      return pInstance_;
   }
private:
   // Создаем новый объект класса Singleton
   // и присваиваем указатель на него переменной pInstance_
   static void Create()
   {
     // Задача: инициализировать переменную pInstance_
     static Singleton theInstance;
     pInstance_=&theInstance;
   }
```

```
    // Вызывается, если обнаружена висячая ссылка
    static void OnDeadReference()
    {
      throw std::runtime_error("Обнаружена висячая ссылка");
    }
    virtual ~Singleton()
    {
      pInstance_ = 0;
      destroyed_ = true;
    }
    // Данные
    Singleton pInstance_;
    bool destroyed_;
    ... заблокированные операторы ...
};

//Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
bool Singleton::destroyed_ = false;
```

Этот код работает! В момент завершения работы приложения вызывается деструктор класса Singleton. Он присваивает переменной pInstance_ значение 0, а переменной destroyed_ — значение true. Если какой-нибудь объект попытается получить доступ к уничтоженному синглтону, поток управления достигнет функции OnDeadReference, которая сгенерирует исключительную ситуацию runtime_error. Это простое и эффективное решение, не требующее больших затрат ресурсов.

6.6. Проблема адресации висячей ссылки (I): феникс

Если описанное выше решение применить к проблеме KDL (Keyboard, Display, Log), результат окажется неутешительным. Если деструктору класса Display понадобится сообщить об ошибке уничтоженному объекту класса Log, функция-член Log::Instance сгенерирует исключительную ситуацию. Если раньше поведение приложения было непредсказуемым, то теперь оно стало неудовлетворительным.

Нам нужно, чтобы доступ к объекту класса Log был постоянным, независимо от того, когда он был создан. В крайнем случае можно было бы создать объект класса Log вновь (после его уничтожения), чтобы получить возможность использовать его для генерации сообщения об ошибках *в любое время*. Эта идея положена в основу шаблона проектирования Phoenix Singleton.

Подобно легендарной птице Феникс, восстающей из пепла, такой синглтон может возникнуть вновь после своего уничтожения. В каждый фиксированный момент времени по-прежнему существует лишь один экземпляр класса Singleton (двух синглтонов одновременно быть не может). Тем не менее при обнаружении висячей ссылки объект этого класса можно создавать вновь. Используя шаблон проектирования Phoenix Singleton, можно легко решить проблему KDL: классы Keyboard и Display остаются обычными синглтонами, а класс Log становится фениксом.

Феникс можно легко реализовать с помощью статической переменной. При обнаружении висячей ссылки новый объект класса Singleton создается под оболочкой старого. (В языке C++ это возможно. Память, выделенная для статических объектов, остается доступной на всем протяжении работы программы.) Разрушение нового объекта также регистрируется функцией atexit. Изменять функцию Instance не нужно. Все модификации касаются только функции OnDearReference.

```
class Singleton
{
   ... как и раньше ...
   void KillPhoenixSingleton(); // Добавлено
};

void Singleton::OnDeadReference()
{
   // Получаем оболочку уничтоженного синглтона
   Create();
   // Теперь указатель pInstance_ ссылается
   // на "пепел" синглтона — ячейки памяти, которые
   // ранее занимал уничтоженный синглтон.
   // На этом же месте создается новый синглтон
   new(pInstance_) Singleton;
   // Ставим новый объект в очередь на уничтожение
   atexit(KillPhoenixSingleton);
   // Изменяем значение переменной destroyed_,
   // поскольку синглтон восстановлен
   destroyed_ = false;
}

void Singleton::KillPhoenixSingleton()
{
   // Снова все превращаем в пепел — вызываем деструктор явно.
   // Переменной pInstance_ присваивается значение 0,
   // а переменной destroyed_ — значение true
   pInstance_->~Singleton();
}
```

Оператор new, используемый в функции OnDeadReference, называется *оператором размещения* (placement new operator). Этот оператор не выделяет память, он лишь создает новый объект и размещает его по указанному адресу, в нашем случае — по адресу, указанному в переменной pInstance_. Интересное обсуждение этого оператора можно найти в книге (Meyers, 1998b).

В приведенном выше описании класса Singleton добавлена новая функция KillPhoenixSingleton. Зная, что для возрождения феникса в программе используется оператор размещения new, компилятор больше не разрушает его, как обычную статическую переменную. Мы создаем его вручную, поэтому и уничтожать его должны сами, для чего и предназначен вызов функции atexit(KillPhoenixSingleton).

Проанализируем поток событий. При выходе из приложения вызывается деструктор класса Singleton. Он устанавливает указатель на нулевой адрес и присваивает переменной destroyed_ значение true. Предположим теперь, что к объекту класса Singleton вновь пытается получить доступ некий глобальный объект. Тогда функция Instance вызовет функцию OnDeadReference, которая реанимирует объект класса Singleton и ставит в очередь вызов функции KillPhoenixSingleton. После этого функция Instance успешно возвращает ссылку на правильный объект класса Singleton. Теперь цикл можно повторять снова.

Феникс гарантирует, что глобальный объект и другие синглтоны в любое время могут обратиться к его правильному экземпляру. Это делает феникс привлекательным средством для создания надежных и всегда доступных объектов, таких как объект класса Log. Если сделать класс Log фениксом, программа всегда будет работать правильно.

6.6.1. Проблемы, связанные с функцией atexit

Если сравнить код, приведенный в предыдущем разделе, с кодом из библиотеки Loki, обнаружится одно отличие. В библиотеке вызов функции atexit окружен директивой препроцессора #ifdef.

```
#ifdef ATEXIT_FIXED
   // Ставим новый объект в очередь на уничтожение
   atexit(KillPhoenixSingleton);
#endif
```

Если в программе нет директивы #define ATEXIT_FIXED, вновь созданный феникс не будет уничтожен. Это приведет к утечке памяти, которой мы стремимся избежать.

Это происходит потому, что в стандарте языка C++ есть досадный пробел. В нем не описано, что случится, если функция регистрируется с помощью функции atexit во время вызова, который является следствием другой регистрации функции atexit.

Чтобы проиллюстрировать эту проблему, напишем короткую тестовую программу.

```
#include <cstdlib>
void Bar()
{
   ...
}
void Foo()
{
   std::atexit(Bar);
}
int main()
{
   std::atexit(Foo);
}
```

Эта маленькая программа регистрирует функцию Foo с помощью функции atexit. Функция atexit выполняет вызов atexit(Bar). Этот случай не описан ни в одном из стандартов языков C и C++. Поскольку функция atexit и разрушение статических переменных тесно связаны друг с другом, при выходе из приложения мы теряем почву под ногами. Стандарты языков C и C++ внутренне противоречивы. Они утверждают, что функция Bar будет вызвана до функции Foo, поскольку функция Bar зарегистрирована последней. Однако она не может вызываться первой, поскольку функция Foo уже была вызвана.

Может показаться, что это слишком академическая проблема. Посмотрим на нее с другой стороны. Написанная выше программа была проверена на трех широко распространенных компиляторах. Ее поведение изменялось от просто неправильного (утечка ресурсов) до полного краха.[2]

На поиск решения этой проблемы уйдет довольно много времени. В данный момент считается, что лучше всего в этой ситуации применять макросы. В других компиляторах

---

[2] Я обсуждал эту проблему как в группе новостей (comp.std.c++), так и по электронной почте со Стивом Кламажем (Steve Klamage), председателем комитета по стандартизации языка C++ (ANSI/ISO C++ Standards Committee). Эта проблема ему хорошо известна, и отчет об этом дефекте уже направлен в соответствующие комитеты. Их можно найти на Web-странице http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/lwg-issues.html#3. К счастью, наиболее приемлемым решением этой проблемы в настоящее время считается реализация класса Singleton, описанная здесь. Даже с функциями, вызываемыми в момент выхода из программы, функция atexit работает по принципу стека, что и требуется.

при вторичном создании феникса в конце концов произойдет утечка памяти. Если компилятор позволяет, можно вставить в программу директиву #define ATEXIT_FIXED перед директивой включения заголовочного файла Singleton.h.

6.7. Проблема адресации висячей ссылки (II): синглтон с заданной продолжительностью жизни

Феникс удобен во многих ситуациях, но имеет массу недостатков. Он нарушает нормальный цикл жизни синглтона, что может создать проблемы для некоторых пользователей. В результате выполнения цикла разрушения-восстановления синглтона его состояние теряется. Программист, связанный с реализацией конкретного синглтона, использующий стратегию феникса (Phoenix strategy), должен уделить особое внимание хранению его состояния между моментами его разрушения и восстановления.

Это особенно досадно, поскольку в ситуациях, подобных задаче KDL, нужно *точно* знать порядок действий. Если объект класса Log создан, он в любом случае должен быть разрушен после объектов классов Keyboard и Display. Иными словами, объект класса Log должен жить дольше, чем объекты классов Keyboard и Display. Нам нужен простой способ управления продолжительностью жизни разных синглтонов, и тогда мы сможем решить задачу KDL, задав для объекта классса Log более высокую продолжительность жизни, чем у объектов классов Keyboard и Display.

Однако это еще не все! Описанная выше проблема относится не только к синглтонам, но и к глобальным объектам вообще. Это позволяет ввести понятие *управления продолжительностью жизни объектов* (longevity control) независимо от концепции синглтона. Чем бóльшую продолжительность жизни имеет *объект*, тем позже он будет уничтожен, причем неважно, является он синглтоном или глобальным объектом, размещенным в динамической памяти.

```
// Класс Singleton
class SomeSingleton { ... };
// Обычный класс
Class SomeClass { ... }

SomeClass* pGlobalObject(new SomeClass);

int main()
{
   SetLongevity(&SomeSingleton().Instance(), 5);
   // Гарантирует, что объект pGlobalObject будет уничтожен
   // после экземпляра класса SomeSingleton
   SetLongevity(pGlobal, 6);
   ...
}
```

Функция SetLongevity получает ссылку на объект *любого* типа и целочисленное значение (продолжительность жизни).

```
// Получает ссылку на объект, размещенный в памяти
// с помощью оператора new, и его продолжительность жизни
template <typename T>
void SetLongevity(T* pDynOject, unsigned int longevity);
```

Функция SetLongevity гарантирует, что объект pDynObject переживет все объекты, имеющие меньшую продолжительность жизни. При выходе из приложения все объек-

ты, зарегистрированные функцией SetLongevity, удаляются из памяти с помощью оператора delete в порядке убывания продолжительности их жизни.

Функцию SetLongevity нельзя применять к объектам, продолжительность жизни которых управляется компилятором, например, к обычным глобальным объектам и автоматическим объектам. Компилятор уже создал код для их уничтожения, и вызов функции SetLongevity для этих объектов может спровоцировать их повторное уничтожение. (Это никогда не приводит ни к чему хорошему.) Функция SetLongevity предназначена для объектов, созданных только с помощью оператора new. Более того, применение функции SetLongevity к этим объектам отменяет вызов оператора delete для их удаления.

В качестве альтернативы можно создать объект для управления зависимостью, который контролировал бы зависимость между объектами. Класс DependencyManager мог бы содержать функцию SetDependence, как показано ниже.

```
class DependencyManager
{
public:
   template <typename T, typename U>
   void SetDependency(T* dependent, U& target);
   ...
};
```

Деструктор класса DependencyManager уничтожал бы объекты по порядку, перед этим разрушая зависимости между ними.

Проектное решение, основанное на применении класса DependencyManager, имеет большой недостаток — оба объекта должны существовать одновременно. Это значит, что при попытке установить зависимость между объектами классов Keyboard и Log, например, придется создать объект класса Log, даже если в дальнейшем он совершенно не нужен.

Для того чтобы избежать этой ловушки, можно установить зависимость между объектами классов Keyboard и Log внутри конструктора класса Log. Однако это создает слишком сильную связь: объект класса Keyboard зависит от определения класса Log (поскольку он использует этот класс), а объект класса Log зависит от определения класса Keyboard (поскольку он задает зависимость между ними). Это нежелательное явление называется *циклической зависимостью* (circular dependency). Оно подробно обсуждается в главе 10.

Вернемся к парадигме продолжительности жизни. Поскольку функция SetLongevity должна осторожно работать с функцией atexit, нужно тщательно определить точную последовательность вызовов деструктора в следующей программе.

```
class SomeClass{ ... }

int main()
{
   // Создаем объект и задаем его продолжительность жизни
   SomeClass* pObj1 = new SomeClass;
   SetLongevity(pObj1, 5);
   // Создаем статический объект, продолжительность жизни
   // которого подчиняется правилам языка C++
   static SomeClass obj2;
   // Создаем другой объект и задаем еще большую
   // продожительность его жизни
   SomeClass* pObj3 = new SomeClass;
   SetLongevity(pObj3, 6);
   // В каком порядке будут уничтожены эти объекты?
```

В функции main определены как объекты, продолжительность жизни которых задается программистом, так и объекты, подчиняющиеся правилам языка C++. Опреде-

лить разумный порядок уничтожения этих трех объектов довольно трудно, поскольку кроме функции atexit у нас нет никаких средств для манипулирования скрытым стеком, предусмотренным системой поддержки выполнения программ.

Тщательный анализ этих ограничений приводит к следующим проектным решениям.

- Каждый вызов функции SetLongevity должен сопровождаться вызовом функции atexit.
- Уничтожение объектов, имеющих меньшую продолжительность жизни, должно предшествовать уничтожению объектов с большей продолжительностью жизни.
- Разрушение объектов, имеющих одинаковую продолжительность жизни, должно следовать правилу языка C++: последним создан — первым уничтожен.

В приведенном выше примере эти правила приводят к следующему порядку уничтожения объектов: *pObj1, obj2, *pObj3. Первый вызов функции SetLongevity сопровождается вызовом функции atexit для уничтожения объекта *pObj3, второй вызов, соответственно, заканчивается вызовом функции atexit для уничтожения объекта *pObj1.

Функция SetLongevity дает программистам возможность управлять продолжительностью жизни объектов и хорошо согласуется с правилами языка C++. Заметим, однако, что, как и многие мощные инструменты, она может оказаться опасной. Следует придерживаться следующего правила: любой объект, использующий объект с заданной продолжительностью жизни, должен иметь меньшую продолжительность жизни, чем используемый объект.

6.8. Реализация синглтонов, имеющих заданную продолжительность жизни

Несмотря на то что спецификация функции SetLongevity завершена, реализация не закончена. Функция SetLongevity поддерживает скрытую *очередь с приоритетами* (priority queue), не связанную с недоступным стеком функции atexit. В свою очередь функция SetLongevity вызывает функцию atexit, всегда передавая ей один и тот же указатель на функцию, которая выталкивает из стека и удаляет один элемент. Все это довольно просто.

Проблема заключается в использовании очереди с приоритетами, которые устанавливаются в соответствии с продолжительностью жизни, передаваемой функции SetLongevity в виде параметра. Для заданной продолжительности жизни очередь функционирует как стек. Уничтожение объектов, имеющих одинаковую продолжительность жизни, осуществляется по правилу "последним пришел, первым ушел". Несмотря на совпадение имен, мы не можем использовать стандартный класс std::priority_queue, поскольку он не устанавливает порядок следования элементов, имеющих одинаковый приоритет.

Элементы данной структуры данных содержат указатели на тип LifetimeTracker. Их интерфейс состоит из виртуального деструктора и оператора сравнения. В производных классах деструктор должен замещаться. (Мы вскоре увидим, какая функция Compare для этого подходит лучше всего.)

```
namespace Private
{
   class LifetimeTracker
   {
   public:
      LifetimeTracker(unsigned int x) : longevity_(x) {}
```

```
    virtual ~LifetimeTracker() = 0;
    friend inline bool Compare(
      unsigned int longevity,
      const LifetimeTracker* p)
    { return p->longevity_ > longevity; }
  private:
    unsigned int longevity_;
  };
  // Необходимое определение
  inline LifetimeTracker::~LifetimeTracker() {}
}
```

Очередь с приоритетами представляет собой динамический массив указателей на функции LifetimeTracker.

```
namespace Private
{
  typedef LifetimeTracker** TrackerArray;
  extern TrackerArray pTrackerArray;
  extern unsigned int elements;
}
```

В программе может существовать лишь один экземпляр типа Tracker. Следовательно, при работе с массивом pTrackerArray возникают все те же проблемы, указанные выше для класса Singleton. Возникает интересная проблема "курицы и яйца": функция SetLongevity должна быть и закрытой, и доступной одновременно. Чтобы решить эту проблему, функция SetLongevity использует для манипуляций с массивом pTrackerArray функции низкого уровня из семейства std::malloc (malloc, realloc и free)[3]. Таким образом, решение проблемы "курицы и яйца" перекладывается на механизм распределения динамической памяти в языке С, гарантирующий правильную работу приложения. Следует отметить, что реализация функции SetLongevity довольно проста. Она создает конкретный объект класса LifetimeTracker, добавляет его в стек и регистрирует вызов функции atexit.

Приведенный ниже код очень важен для обобщения. Он вводит в рассмотрение функторы, предназначенные для *уничтожения* отслеживаемых объектов. Это позволяет не использовать оператор delete для удаления объекта из динамической памяти, поскольку может оказаться, что объект находится в альтернативной куче или где-то еще. По умолчанию механизм уничтожения объекта представляет собой указатель на функцию, вызывающую оператор delete. Эта функция называется Delete, а ее шаблонный параметр задает тип удаляемого объекта.

```
// Вспомогательная функция для удаления объектов
template <typename T>
struct Deleter
{
  static void Delete(T* pObj)
  { delete pObj; }
};
// Конкретный объект класса LifetimeTracker для объектов типа T
template <typename T, typename Destroyer>
class ConcreteLifetimeTracker : public LifetimeTracker
{
```

---

[3] Фактически функция SetLongevity использует только функцию std::realloc, заменяющую собой и функцию malloc, и функцию free. Если вызвать ее с нулевым указателем, она ведет себя как функция std::malloc, а если задать нулевой размер, она имитирует работу функции std::free.

```
public:
   ConcreteLifetimeTracker(T* p,
      unsigned int longevity,
      Destroyer d)
      :LifetimeTracker(longevity),
      pTracked_(p),
      destroyer_(d)
      {}
   ~ConcreteLifetimeTracker()
   {
     destroyer_(pTracked_);
   }
private:
   T* pTracked_;
   Destroyer destroyer_;
};
  void AtExitFn(); // Необходимое объявление

template <typename T, typename Destroyer>
void SetLongevity(T* pDynObject, insigned int longevity,
   Destroyer d = Private::Deleter<T>::Delete)
{
   TrackerArray pNewArray = static_cast<TrackerArray>(
      std::realloc(pTrackerArray, elements + 1));
   if (!pNewArray) throw std::bad_alloc();
   LifetimeTracker* p = new ConcreteLifetimeTracker
      <T, Destroyer>(pDynObject, longevity, d);
   pTrackerArray = pNewArray;
   TrackerArray pos = std::upper_bound(
      pTrackerArray, pTrackerArray + elements,
      longevity, Compare);
   std::copy_backward(pos, pTrackerArray + elements,
      pTrackerArray + elements + 1);
   *pos = p;
   ++elements;
   std::atexit(AtExitFn);
}
```

Использование функций std::upper_bound и std::copy_backward намного облегчает чтение и понимание этого нетривиального кода. Описанная выше функция вставляет вновь созданный указатель на объект класса ConcreteLifetimeTracker в упорядоченный массив, на который ссылается указатель pTrackerArray, сохраняет порядок следования его элементов, а также обрабатывает ошибки и возникающие исключительные ситуации.

Теперь цель функции LifetimeTracker::Compare проясняется. Массив, на который ссылается указатель pTrackerQueue, упорядочен в соответствии с продолжительностью жизни объектов. Объекты с наибольшей продолжительностью жизни находятся в начале массива. Объекты с одинаковой продолжительностью жизни следуют в соответствии с очередностью их вставки.

Функция AtExitFn выталкивает объект с наименьшей продолжительностью жизни (т.е. последний элемент массива) и удаляет его. Удаление указателя на объект класса LifetimeTracker приводит к вызову деструктора класса ConcreteLifetimeTracker, который в свою очередь удаляет объект.

```
static void AtExitFn()
{
   assert(elements > 0 && pTrackerArray != 0);
   // Выбираем элемент, находящийся на вершине стека
   LifetimeTracker* pTop = pTrackerArray[elements - 1];
```

```
    // Удаляем этот объект из стека.
    // Ошибки не проверяются — функция realloc,
    // примененная к меньшему размеру памяти,
    // всегда работает правильно
    pTrackerArray = (TrackerArray*)std::realloc(pTrackerArray,
    --elements);
    // Уничтожаем элемент
    delete pTop;
}
```

Создавая функцию AtExitFn, следует быть внимательным. Она должна выталкивать из стека элемент, находящийся на вершине, и удалять его. В свою очередь деструктор удаляемого элемента ликвидирует управляемый им объект. Проблема заключается в том, что функция AtExitFn должна вытолкнуть объект из вершины стека *до* его удаления, поскольку разрушение одного объекта может повлечь за собой создание другого, который будет затолкнут обратно в стек. Несмотря на то что это выглядит довольно необычно, именно так развиваются события, когда деструктор класса Keyboard пытается использовать объект класса Log.

По умолчанию код скрывает структуры данных, и функция AtExitFn находится в пространстве имен Private. Пользователи могут любоваться лишь вершиной айсберга — функцией SetLongevity.

Синглтоны с заданной продолжительностью жизни могут использовать функцию SetLongevity следующим образом.

```
class Log
{
public:
    static void Create()
    {
        // Создаем экземпляр
        pInstance_ = new Log;
        // Эти строки добавлены
        SetLongevity(*this, longevity_);
    }
    // Остальная часть реализации пропущена.
    // Функция Log::Instance определяется, как и прежде.
private:
    // Определяем фиксированную продолжительность жизни
    static const unsigned int longevity_ = 2;
    static Log* pInstance_;
};
```

Если реализовать классы Keyboard и Display, следуя описанному выше подходу, но задать продолжительность жизни их объектов равной 1, то объект класса Log их обязательно переживет. Решает ли это проблему KDL? Что если приложение использует несколько потоков?

6.9. Продолжительность жизни объектов в многопоточной среде

Синглтоны должны работать и с потоками тоже. Допустим, наше приложение только что начало работу, и два потока имеют доступ к приведенному ниже синглтону.

```
Singleton& Singleton::Instance()
{
    if(!pInstance)                    // 1
```

```
    {
        pInstance_ = new Singleton;  // 2
    }
    return *pInstance_;              // 3
}
```

Первый поток входит в функцию Instance и проверяет условие оператора if. Поскольку поток входит в функцию впервые, значение переменной pInstance_ равно 0. В таком случае поток управления достигает строки с комментарием //2 и готовится вызвать оператор new. Планировщик заданий операционной системы может прервать первый поток в этой точке и передать управление другому потоку.

Второй поток вызывает функцию Singleton::Instance() и обнаруживает, что значение переменной pInstance_ также равно нулю, поскольку первый поток ее еще не менял. До сих пор первый поток только *проверял* значение переменной pInstance_. Теперь второй поток вызывает оператор new, присваивает переменной pInstance_ некий адрес и покидает функцию.

К несчастью, первый поток снова приходит в сознание, вспоминает, что осталось выполнить лишь строку с комментарием //2, изменяет значение переменной pInstance_ и выходит из функции. Когда пыль рассеивается, оказывается, что вместо одного объекта класса Singleton созданы два, причем один из них очевидно лишний. В каждом потоке хранится по одному объекту класса Singleton, и приложение погружается в хаос. И это только одна из возможных ситуаций! А что будет, если к синглтону имеют доступ несколько потоков? (Представьте себе процесс отладки такой программы!)

Опытные программисты, разрабатывающие многопоточные приложения, узнают здесь классическое состязание (race situation). Следует быть готовым к тому, что шаблон проектирования Singleton столкнется с потоками. Синглтон относится к глобальным ресурсам совместного использования и должен участвовать в состязании с другими объектами, решая проблемы нескольких потоков.

6.9.1. Шаблон блокировки с двойной проверкой

Всестороннее обсуждение многопоточных синглтонов впервые было проведено Дугласом Шмидтом (Douglas Schmidt, 1996). В этой же статье было описано очень остроумное решение, названное шаблоном Double-Checked Locking (блокировка с двойной проверкой), предложенное Дугласом Шмидтом и Тимом Харрисоном (Tim Harrison).

Очевидное решение существует, но выглядит непривлекательно.

```
Singleton& Singleton::Instance()
{
    // mutex_ — объект мьютекса.
    // Мьютексом управляет объект класса Lock
    Lock guard(mutex_);
    if (!pInstance_)
    {
        pInstance_ = new Singleton;
    }
    return *pInstance_;
}
```

Класс Lock — это классический обработчик мьютексов (детали описаны в приложении). Конструктор класса Lock захватывает мьютекс, а деструктор освобождает его. Пока мьютекс mutex_ захвачен, другие потоки, пытающиеся завладеть им, ожидают своей очереди.

Это освобождает нас от состязания: пока поток присвоен объекту pInstance_, остальные останавливаются в конструкторе guard. Когда другой поток пытается захватить блокировку, он обнаруживает уже проинициализированную переменную pInstance_, и все проходит гладко.

Однако правильное решение не всегда оказывается приемлемым. Неудобство заключается в недостатке эффективности. Каждый вызов функции-члена Instance влечет за собой блокировку и освобождение объекта синхронизации, хотя состязание за ресурсы возникает, образно говоря, один раз в жизни. Эти операции обычно очень затратны. Их стоимость намного превышает стоимость выполнения простой проверки if(!pInstance). (В современных системах время, затрачиваемое на проверку и ветвление, обычно отличается от времени, уходящего на блокировку критических разделов, на несколько порядков.)

Для того чтобы избежать дополнительных затрат, можно было бы предложить следующее решение.

```
Singleton& Singleton::Instance()
{
   if (!pInstance_)
   {
      Lock guard(mutex_);
      pInstance_ = new Singleton;
   }
   return *pInstance_;
}
```

Теперь дополнительных затрат нет, однако осталось состязание за ресурсы. Первый поток проходит проверку if, однако прямо перед входом в синхронизированную часть кода планировщик заданий операционной системы прерывает его и передает управление другому потоку. Второй поток также проходит проверку if (и, естественно, обнаруживает нулевой указатель), входит в синхронизированную часть кода и выполняет ее. После повторной активизации первый поток также входит в синхронизированную часть, но слишком поздно — снова создаются два объекта класса Singleton.

Кажется, что эта головоломка не имеет решения, но оказывается, что существует очень простое и элегантное решение. Оно называется шаблоном Double-Checked Locking (блокировка с двойной проверкой).

Идея проста: проверяем условие, входим в синхронизированный код, а затем проверяем это условие снова. Указатель оказывается либо уже проинициализированным, либо нулевым. Ниже приведен код, позволяющий разобраться и оценить преимущества шаблона блокировки с двойной проверкой. В нем действительно проявляется красота программирования.

```
Singleton& Singleton::Instance()
{
   if (!pInstance)              // 1
   {                            // 2
      Guard myGuard(lock_);     // 3
      if (!pInstance_)          // 4
      {
         pInstance_ = new Singleton;
      }
   }
   return *pInstance_;
}
```

Допустим, что поток управления входит в зону неопределенности (строка 2). В эту точку могут войти сразу несколько потоков. Однако в синхронизированный раздел входит только один поток. В строке 3 неопределенности вообще нет. Здесь все ясно: указатель либо полностью проинициализирован, либо не проинициализирован вообще. Первый поток, который входит в этот раздел, инициализирует переменную, а все остальные не пройдут проверки на строке 4 и ничего не создадут.

Первая проверка быстрая и грубая. Если объект класса Singleton доступен, вы его получаете. Если нет, необходимы дальнейшие исследования. Вторая проверка медленна и точна: она сообщает, действительно ли проинициализирован синглтон, или это должен сделать поток. В этом и заключается суть блокировки с двойной проверкой. Теперь мы получаем большое преимущество: скорость доступа к синглтону высока настолько, насколько позволяет сам объект. Однако во время создания объекта класса Singleton состязание больше не возникает. И все же...

Программисты, имеющие очень большой опыт работы с потоками, знают, что даже шаблон блокировки с двойной проверкой, правильный на бумаге, не всегда хорошо работает на практике. В некоторых симметричных мультипроцессорных системах (с так называемой релаксированной моделью памяти) порции информации загружаются в память одновременно, а не последовательно. Эти порции записываются по возрастанию адресов, а не в хронологическом порядке. Из-за такого способа упорядочения записей память, просматриваемая одним процессором, может выглядеть так, будто порядок операций, выполненных другим процессором, был неправильным. Например, присваивание значения переменной pInstance_ может выполняться до завершения полной инициализации объекта класса Singleton! Таким образом, увы, шаблон блокировки с двойной проверкой оказался непригодным для таких систем.

В заключение следует заметить, что перед реализацией шаблона Double-Checked Locking нужно просмотреть документацию компилятора. (Тогда его можно назвать шаблоном Triple-Checked Locking (блокировка с тремя проверками).) Обычно операционная система предоставляет альтернативные, машинно-зависимые средства решения проблемы параллелизма, например, барьеры, упорядочивающие доступ к памяти (memory barriers). По крайней мере, поставьте перед переменной pInstance_ спецификатор volatile. Хороший компилятор должен генерировать правильный код вокруг таких объектов.

6.10. Сборка

В этой главе обсуждаются разные реализации класса Singleton, выявляются их относительные преимущества и недостатки. Не следует думать, что в результате мы сможем прийти к универсальному решению, поскольку каждая задача предъявляет к реализации класса Singleton свои требования.

Шаблонный класс SingletonHolder, определенный в библиотеке Loki, представляет собой контейнер для синглтонов, позволяющий применить шаблон проектирования Singleton. Следуя шаблону проектирования, основанному на применении стратегий (глава 1), класс SingletonHolder разработан как специализированный контейнер для объектов класса Singleton, определенного пользователем. При использовании класса SingletonHolder программист получает все необходимые функциональные возможности и может создавать свой собственный код. В крайнем случае придется все переделать заново (поэтому этот случай и называют крайним).

В этой главе рассмотрено несколько тем, практически не связанных друг с другом. Как же теперь реализовать класс Singleton, не раздувая размер программы? Для этого нужно

тщательно разложить шаблон Singleton на стратегии, как показано в главе 1. Разложив класс SingletonHolder на несколько стратегий, можно реализовать *все* варианты, рассмотренные выше, с помощью кода, состоящего из небольшого количества строк. Используя конкретизацию шаблонов, можно отобрать желательные свойства и пренебречь ненужными. Это очень важно: реализация класса Singleton не универсальна. Используются лишь те свойства, которые в итоге будут включены в сгенерированный код. Кроме того, реализация оставляет возможности для изменения и расширений.

6.10.1. Разложение класса SingletonHolder на стратегии

Начнем с выделения стратегий из описанной выше реализации. В ней можно идентифицировать вопросы, связанные с созданием объекта, его продолжительностью жизни и потоками. Это три наиболее важных момента в разработке синглтонов. Рассмотрим соответствующие стратегии.

1. Стратегия Creation. Синглтон можно создать разными способами. Обычно для создания объекта стратегия Creation использует оператор new. Выделение процесса создания объекта в виде отдельной стратегии существенно, поскольку это позволяет создавать полиморфные объекты.
2. Стратегия Lifetime. Различаются следующие стратегии продолжительности жизни объектов.
   2.1. Правила языка C++ — последним создан, первым уничтожен.
   2.2. Феникс.
   2.3. Синглтон с заданной продолжительностью жизни.
   2.4. Бессмертный синглтон (объект, который никогда не уничтожается).
3. Стратегия ThreadingModel. Синглтон может работать в режиме одного потока, в стандартном многопоточном режиме (с мьютексами и блокировкой с двойной проверкой) или использовать платформно-зависимую потоковую модель.

Все реализации класса Singleton должны гарантировать уникальность объекта. Это условие не влияет на выбор стратегии, поскольку его невыполнение нарушает определение синглтона.

6.10.2. Требования, предъявляемые к стратегиям класса SingletonHolder

Определим необходимые ограничения, накладываемые классом SingletonHolder на свои стратегии.

Стратегия Creation должна создавать и уничтожать объекты, следовательно, она должна содержать две соответствующие функции. Таким образом, если класс Creator<T> согласован со стратегией Creation, он должен содержать вызовы следующих функций.

```
T* pObj = Creator<T>::Create();
Creator<T>::Destroy(pObj);
```

Обратите внимание на то, что функции Create и Destroy должны быть статическими членами класса Creator. Класс Singleton не содержит объект, имеющий тип Creator, — это не позволило бы решить проблемы, связанные с его продолжительностью жизни.

Стратегия Lifetime, по существу, должна планировать разрушение объекта класса Singleton, созданного стратегией Creation. Функциональные возможности стратегии Lifetime выражаются в ее способности разрушать объект класса Singleton в заданный момент времени. Кроме того, стратегия Lifetime решает, какие действия

следует предпринять, если приложение нарушает правила языка C++, определяющие продолжительность жизни синглтона.

- Если синглтон нужно уничтожить, следуя правилам языка C++, стратегия Lifetime применяет механизм, аналогичный функции atexit.
- Для феникса стратегия Lifetime продолжает использовать механизм, аналогичный функции atexit, но допускает восстановление синглтона.
- Для синглтона с заданной продолжительностью жизни стратегия Lifetime вызывает функцию SetLongevity, описанную в разделах 6.7 и 6.8.
- При неограниченной продолжительности жизни синглтона стратегия Lifetime не предпринимает никаких действий.

В заключение отметим, что стратегия Lifetime содержит две функции: ScheduleDestruction, задающую время уничтожения объекта, и OnDeadReference, регламентирующую поведение программы при обнаружении висячей ссылки.

Предположим, что стратегия Lifetime реализуется классом Lifetime<T>. Тогда имеют смысл следующие выражения.

```
void (*pDestructionFunction)();
...
Lifetime<T>::ScheduleDestruction(pDestructionFunction);
Lifetime<T>::OnDeadReference();
```

Функция-член ScheduleDestruction получает указатель на функцию, уничтожающую объект. Таким образом, стратегию Lifetime можно использовать вместе со стратегией Creation. Не забывайте, что стратегия Lifetime не связана с методами уничтожения объектов, которые относятся к стратегии Creation. Единственное предназначение стратегии Lifetime — определять, *когда* объект должен быть уничтожен.

Функция OnDeadReference генерирует исключительные ситуации всегда, за исключением ситуаций, в которых задействован феникс. В последнем случае стратегия не предусматривает никаких действий.

Стратегия ThreadingModel описана в приложении. Класс SingletonHolder поддерживает блокировку только на уровне классов, но не на уровне объектов. По этой причине в любой момент времени существует лишь один синглтон.

6.10.3. Сборка класса SingletonHolder

Начнем определение шаблонного класса SingletonHolder. Как указывалось в главе 1, для каждой стратегии предназначен отдельный шаблонный параметр. Кроме того, мы предусмотрим шаблонный параметр T, определяющий разновидность синглтона. Шаблонный класс SingletonHolder сам по себе не определяет синглтон. Он лишь задает его поведение и способы управления им с помощью уже существующего класса Singleton.

```
template
<
   class T,
   template <class> class CreationPolicy = CreateUsingNew,
   template <class> class LifetimePolicy = DeafaultLifetime,
   template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder
{
public:
   static T& Instance();
```

```
private:
   // Вспомогательные функции
   static void DestroingSingleton();
   // Защита
   SingletonHolder();
   ...
   // Данные
   typedef ThreadingModel<T>::VolatileType InstanceType;
   static InstanceType* pInstance_;
   static bool destroyed_;
};
```

Вопреки ожиданиям, переменная экземпляра не относится к типу T*. На самом деле она имеет тип ThreadingModel<T>::VolatileType*. Тип ThreadingModel<T>::VolatileType расширяет тип T или тип volatile T в зависимости от фактически применяемой модели потоков. Спецификатор volatile применяется к типам для того, чтобы сообщить компилятору, что значение данного типа может изменяться несколькими потоками. Зная это, компилятор не станет выполнять некоторые виды оптимизации (например, запись значений во внутренние регистры), поскольку это может привести к ошибочной работе потоков. В целях безопасности можно было бы объявить переменную pInstance_ с типом volatile T*. Это сработает при использовании многопотокового кода (этот факт следует предварительно проверить по документации), но бесполезно в программе с одним потоком.

С другой стороны, в модели с одним потоком следует стремиться к оптимизации программы, так что тип T* был бы для переменной pInstance_ наилучшим выбором. По этой причине тип для переменной pInstance_ выбирается стратегией ThreadingModel. Если эта стратегия является однопоточной, определяется тип VolatileType.

```
template <class T> class SingleThreaded
{
   ...
public:
   typedef T VolatileType;
};
```

Многопоточная стратегия связывала бы параметр T с типом volatile. Детали многопоточных моделей описаны в приложении.

Определим теперь функцию-член Instance, в которой объединяются все три стратегии.

```
template <...>
T& SingletonHolder<...>::Instance()
{
   if (!pInstance_)
   {
      typename ThreadingModel<T>::Lock guard;
      if (!pInstance_)
      {
         if (destroyed_)
         {
            LifetimePolicy<T>::OnDeadReference();
            destroyed_ = false;
         }
         pInstance_ = CreationPolicy<T>::Create();
         LifetimePolicy<T>::ScheduleCall(&DestroySingleton);
      }
   }
   return *pInstance_;
}
```

Функция Instance является единственным открытым членом класса SingletonHolder. Она представляет собой оболочку классов CreationPolicy, LifetimePolicy и ThreadingModel. Класс ThreadingModel<T> содержит внутренний класс Lock. На протяжении жизни объекта класса Lock все другие потоки, пытающиеся создать объект этого типа, блокируются. (Детали описаны в приложении.)

Функция DestroySingleton просто разрушает объект класса Singleton, очищает занятую память и присваивает переменной destroyed_ значение true. Класс SingletonHolder никогда не вызывает функцию DestroySingleton, а лишь передает ее адрес функции-члену LifetimePolicy<T>::ScheduleDestruction.

```
template <...>
void SingletonHolder<...>::DestroySingleton()
{
   assert(!destroyed_);
   CreationPolicy<T>::Destroy(pInstance_);
   pInstance_ = 0;
   destroyed_ = true;
}
```

Класс SingletonHolder передает переменную pInstance_ и адрес функции DestroySingleton классу LifetimePolicy<T>, предоставляя ему информацию о поведении объекта: подчиняется он правилам языка C++ или является фениксом, имеет заданную продолжительность жизни или бессмертен.

1. *Правила C++.* Функция LifetimePolicy<T>::ScheduleDesctruction вызывает функцию atexit, передавая ей адрес функции DestroySingleton. Функция OnDeadReference генерирует исключительную ситуацию std::logic_error.
2. *Феникс.* Совпадает с предыдущей стратегией, только функция OnDeadReference не генерирует исключительную ситуацию. Поток управления класса SingletonHolder продолжает выполнение программы и воссоздает объект вновь.
3. *Синглтон с заданной продолжительностью жизни.* Функция LifetimePolicy<T>::ScheduleDesctruction вызывает функцию SetLongevity(getLongevity(pInstance)).
4. *Бессмертный синглтон.* Функция LifetimePolicy<T>::ScheduleDesctruction не выполняет никаких действий.

Класс SingletonHolder решает проблему висячей ссылки в соответствии со стратегией LifetimePolicy. Она очень проста: если функция SinlgetonHolder::Instance обнаруживает висячую ссылку, она вызывает функцию LifetimePolicy::OnDeadReference. Если функция OnDeadReference возвращает управление, функция Instance воссоздает новый экземпляр. В заключение функция OnDeadReference должна сгенерировать исключительную ситуацию или прекратить выполнение программы, если синглтон не является фениксом.

Вот и вся реализация класса SingletonHolder. Разумеется, теперь основная часть работы перекладывается на три стратегии.

6.10.4. Реализации стратегий

Разложить класс на стратегии трудно, зато потом их легко реализовывать. Рассмотрим классы, реализующие стратегии для распространенных разновидностей синглтонов. В табл. 6.1 приведены классы стратегий для класса SingletonHolder. Классы стратегий, выделенные курсивом, являются шаблонными параметрами, задаваемыми по умолчанию.

Таблица 6.1. Стратегии класса SingletonHolder

Стратегия	Шаблонный класс	Комментарии
Creation	*CreateUsingNew*	Создает объект с помощью оператора new и конструктора по умолчанию
	CreateUsingMalloc	Создает объект с помощью функции std::malloc и конструктора по умолчанию
	CreateStatic	Создает объект в статической памяти
Lifetime	*DefaultLifeTime*	Управляет продолжительностью жизни объекта в соответствии с правилами языка C++. Для выполнения задания вызывает функцию atexit
	PhoenixSingleton	Выполняет то же самое, что и класс DefaultLifetime, но допускает воссоздание объекта класса Singleton
	SingletonWithLongevity	Задает продолжительность жизни объекта класса Singleton. Предполагается, что в пространстве имен существует функция GetLongevity, которая, будучи вызванной с параметром pInstance_, возвращает продолжительность жизни синглтона
	NoDestroy	Не допускает уничтожения объекта класса Singleton
ThreadingModel	*SingleThreaded*	Детали описаны в приложении
	ClassLevelLockable	

Осталось только узнать, как использовать и расширять этот маленький, но довольно мощный шаблонный класс SingletonHolder.

6.11. Работа с классом SingletonHodler

Шаблонный класс SingletonHolder не требует от приложения специфических функциональных возможностей. Он просто предоставляет специальные средства для работы с синглтонами в других классах — в нашем коде они обозначены буквой T. Класс T мы будем называть *клиентским* (client class).

В клиентском классе следует предусмотреть все меры для предотвращения непреднамеренного создания и уничтожения объектов: конструктор по умолчанию, конструктор копирования, оператор присваивания, деструктор и оператор взятия адреса должны быть закрытыми.

Предприняв эти меры, нужно объявить дружественным класс Creator. Превентивные меры и объявление friend — вот и все изменения, которые необходимо внести в класс, работающий с классом SingletonHolder. Отметим, что все эти изменения совершенно не являются обязательными и представляют собой компромисс между неудобствами настройки существующего кода и риском возникновения фиктивных объектов.

Проектные решения, касающиеся работы со специальной реализацией класса Singleton, отражаются в определении типа, как показано ниже. Передавая признаки и

опции при вызове некоторой функции, вы передаете их определению типа, выбирая соответствующее поведение объектов.

```
class A { ... }
typedef SingletonHolder<A, CreateUsingNew> SingleA;
// Теперь можно использовать функцию SingleA::Instance()
```

Описать синглтон, возвращающий объект производного класса, довольно просто. Для этого достаточно внести изменения в класс стратегии Creator.

```
class A { ... };
class Derived : public A { ... };

template <class T> struct MyCreator : public CreateUsingNew<T>
{
   static T* Create()
   {
      return new Derived;
   }
};

typedef SingletonHolder<A, StaticAllocator, MyCreator> SingleA;
```

Кроме того, можно задать параметры конструктора, использовать другую стратегию распределения памяти или настроить класс Singleton на каждую отдельную стратегию. Это позволяет точно настроить класс Singleton, сохранив все его функциональные возможности, предусмотренные по умолчанию.

Класс стратегии SingletonWithLongevity полагается на определение функции GetLongevity, находящейся в пространстве имен. Ее определение может выглядеть следующим образом.

```
inline unsigned int GetLongevity(A*) { return 5; }
```

Это нужно, только если вы используете класс SingleWithLongevity в определении типа SingleA.

Наши экспериментальные реализации испытывались на задаче KDL. Теперь она решается с помощью шаблонного класса Singleton. Разумеется, эти определения должны находиться в соответствующих заголовочных файлах.

```
class KeyboardImpl { ... };
class DisplayImpl { ... };
class LogImpl { ... };
  ...
inline unsigned int GetLongevity(KeyboardImpl*) { return 1; }
inline unsigned int GetLongevity(DisplayImpl*) { return 1; }
// Объекты класс Log живут дольше
inline unsigned int GetLongevity(LogImpl*) { return 2; }

typedef SingletonHolder<KeyboardImpl, SingletonWithLongevity> Keyboard;
typedef SingletonHolder<DisplayImpl, SingletonWithLongevity> Display;
typedef SingletonHolder<LogImpl, SingletonWithLongevity> Log;
```

Учитывая сложность задачи, описанное решение является достаточно простым и очевидным.

6.12. Резюме

В начале главы была описана наиболее популярная реализация класса Singleton на языке C++. Предотвратить тиражирование синглтонов довольно просто, поскольку

для этого существует хорошая языковая поддержка. Сложнее всего управлять продолжительностью жизни синглтона, особенно процессом его уничтожения.

Распознать попытку доступа к уничтоженному синглтону просто. Для этого не требуются дополнительные ресурсы. Это распознавание должно быть неотъемлемой частью реализации класса `Singleton`.

В главе рассмотрены четыре основные вариации на эту тему: синглтон, управляемый компилятором, феникс, синглтон с заданной продолжительностью жизни и бессмертный синглтон. У каждого из них есть свои преимущества и недостатки.

Шаблон проектирования `Singleton` тесно связан с моделями потоков. Шаблон `Double-Checking Locking` позволяет создать синглтоны, гарантирующие безопасную работу потоков.

В заключении перечислены и классифицированы основные стратегии и выполнена соответствующая декомпозиция класса `Singleton`. В классе `Singleton` выделены три основные стратегии: `Creation`, `Lifetime` и `ThreadingModel`. Эти стратегии сопряжены в определении класса `SingletonHolder` с четырьмя параметрами (клиентский тип плюс три стратегии), которые покрывают все возможные комбинации проектных решений.

6.13. Краткое описание шаблонного класса SingletonHolder

- Определение класса `SingletonHolder` имеет следующий вид.

```
  template <
     class T,
     template <class> class CreationPolicy = CreateUsingNew,
     template <class> class LifetimePlicy = DefaultLifetime,
     template <class> class ThreadingModel = SingleThreaded
>
class SingletonHolder;
```

- Шаблонный класс `SingletonHolder` конкретизируется путем передачи клиентского класса в качестве первого шаблонного параметра. Различные проектные варианты выбираются с помощью комбинирования четырех остальных параметров.

```
class MyClass { ... };
typedef Singleton<MyClass, CreateStatic>
   MySingleClass;
```

- Нужно определить конструктор по умолчанию или использовать для создания синглтона класс, отличающийся от стратегии `Creator`.

- Готовые реализации трех основных стратегий описаны в табл. 6.1. При необходимости к ним можно добавить свои собственные классы стратегий, ориентированные на новые ограничения.

7

Интеллектуальные указатели

Интеллектуальным указателям посвящены миллионы строк кода и сотни книг. Благодаря тому, что в интеллектуальных указателях переплелись многие синтаксические и семантические проблемы, они стали одной из самых популярных, интересных и мощных идиом языка C++. В этой главе обсуждаются все аспекты интеллектуальных указателей (от простейших до самых сложных), а также наиболее распространенные ошибки, совершаемые программистами при их реализации.

Интеллектуальные указатели — это объекты языка C++, имитирующие обычные указатели с помощью реализации оператора -> и унарного оператора *. Кроме того, они часто скрытно выполняют полезные задания (например, осуществляют управление памятью или блокировку), освобождая приложение от детального контроля за объектами, на которые они ссылаются.

В главе не только обсуждаются интеллектуальные указатели, но и реализован шаблонный класс `SmartPtr`. Этот класс разработан на основе стратегий (глава 1). Он безопасен, эффективен и легок в использовании.

Здесь рассмотрены следующие вопросы.

- Преимущества и недостатки интеллектуальных указателей.
- Стратегии управления владением.
- Неявные преобразования.
- Проверки и сравнения.
- Многопоточность.

В главе описан обобщенный шаблонный класс `SmartPtr`. В каждом разделе рассматривается отдельная проблема, связанная с реализацией этого класса, а полная сборка класса выполняется в конце главы. Программисту мало знать, как устроен шаблонный класс `SmartPtr`, нужно еще уметь использовать, изменять и расширять его в своих программах.

7.1. Сто первое описание интеллектуальных указателей

Что такое интеллектуальный указатель? Это класс, имитирующий синтаксис и семантику обычного указателя и выполняющий много другой полезной работы. Поскольку интеллектуальные указатели, ссылающиеся на объекты разных типов, имеют много общего, они почти всегда реализуются в виде шаблонов.

```
template <class T>
class SmartPtr
```

```
{
public:
   explicit SmartPtr(T* pointee) : pointee_(pointee);
   SmartPtr& operator=(const SmartPtr& other);
   ~SmartPtr();
   T& operator*() const
   {
      ...
      return *pointee_;
   }
   T* operator->() const
   {
      ...
      return pointee_;
   }
private:
   T* pointee_;
   ...
};
```

Шаблон SmartPtr<T> хранит указатель на тип T в переменной-члене pointee_. Именно так поступает большинство интеллектуальных указателей. В некоторых случаях в этих классах предусматривается некоторая дополнительная обработка данных, а сам указатель вычисляется на лету.

Синтаксис и семантика обычных указателей имитируются следующими операторами шаблонного класса SmartPtr.

```
class Widget
{
public:
   void Fun();
};
SmartPtr<Widget> sp(new Widget);
sp->Fun();
(*sp).Fun();
```

Помимо определения класса sp, класс SmartPtr ничем не отличается от обычного указателя. В этом и заключается существо интеллектуального указателя: обычные указатели можно заменить интеллектуальными, не внося коренных изменений в программу. Таким образом, можно легко получить дополнительные преимущества. Возможность практически не изменять код при использовании интеллектуальных указателей в больших приложениях — очень привлекательное и важное свойство. Однако, как мы увидим ниже, не все так просто.

7.2. Особенности интеллектуальных указателей

Может возникнуть вопрос: "Зачем нужны интеллектуальные указатели?". Какую выгоду можно извлечь, заменив обычные указатели интеллектуальными? Все очень просто. Интеллектуальные указатели имеют семантику значений, а простые указатели — нет.

Объект, имеющий семантику значений, — это объект, который можно *копировать* и *присваивать*. Числа типа int — прекрасный пример. Их можно свободно создавать, копировать и изменять. Указатель, который применяется для перемещения по буферу, также имеет семантику значений — он инициализируется адресом первой ячейки буфера и перемещается по нему вперед, пока не достигнет конца. По пути значения этого указателя можно копировать в другие переменные, сохраняя промежуточные результаты.

Однако работать с указателями на объекты, созданные с помощью оператора new, нужно совсем иначе. Допустим, мы написали следующее выражение.

```
Widget* p = new Widget;
```

В этом случае переменная p не только указывает на объект класса Widget, размещенный в памяти, но и *владеет* им. Это значит, что при выполнении оператора delete p соответствующий объект класса Widget будет уничтожен, а выделенная для него память освобождена. Таким образом, приведенный ниже код окажется неверным.

```
Widget* p = new Widget;
p = 0; // Присваиваем указателю p нечто иное
```

Это происходит потому, что мы теряем владение объектом, на который ранее ссылался указатель p, и не имеем возможности вернуть его. Это приводит к утечке ресурсов.

Более того, при копировании указателя p в другую переменную компилятор не передает ей автоматически владение участком памяти, на который ссылается указатель. Вместо этого в программе появляются два указателя, содержащих адрес одного и того же объекта. Это нежелательно, поскольку может привести к повторному удалению уже удаленного объекта (со всеми вытекающими катастрофическими последствиями). Следовательно, указатели на динамически размещаемые объекты *не должны* иметь семантики значений — их нельзя копировать и присваивать как попало.

Вот здесь-то и нужны интеллектуальные указатели. Большинство интеллектуальных указателей, помимо имитации простых указателей, *управляют владением* объектами, на которые они ссылаются. Они распознают изменения в правах владения, а их деструкторы освобождают память, следуя тщательно продуманной стратегии. Во многих интеллектуальных указателях содержится достаточно информации, чтобы правильно освободить память, занятую объектом, на который они ссылаются.

Управление владением осуществляется разными способами в зависимости от решаемой задачи. Некоторые интеллектуальные указатели автоматически передают права на владение объектом: после копирования исходному указателю присваивается нулевой адрес, а результирующему — адрес объекта и права на владение. Именно такая стратегия реализована в стандартном интеллектуальном указателе std::auto_ptr. Другие интеллектуальные указатели используют подсчет ссылок: они отслеживают общее количество интеллектуальных указателей, ссылающихся на один и тот же объект, и, когда счетчик обнуляется, удаляют его. В заключение заметим, что существуют интеллектуальные указатели, при копировании которых создается дубликат объекта.

Владение объектом — важное свойство интеллектуальных указателей. Управляя правами владения, интеллектуальные указатели гарантируют целостность и полноценную семантику значений. Поскольку права владения связаны в основном с созданием, копированием и уничтожением интеллектуальных указателей, соответствующие функции-члены наиболее важны.

В следующих разделах мы рассмотрим разные аспекты разработки и реализации интеллектуальных указателей. Нашей целью будет максимально полная имитация простых указателей, но не будем забывать, что интеллектуальный указатель — более широкое понятие.

Между интеллектуальными и обычными указателями существует тонкая грань, отделяющая полный контроль от хаоса. Мы увидим, что добавление на первый взгляд полезных свойств иногда оказывается весьма опасным. При реализации хороших интеллектуальных указателей следует соблюдать точный баланс.

7.3. Хранение интеллектуальных указателей

Для начала зададимся фундаментальным вопросом: "Должна ли переменная pointee_ иметь тип T*?". Если нет, то какой тип она может иметь? В обобщенном программировании обязательно нужно задавать себе такие вопросы. Каждый тип, встроенный в обобщенный код, снижает степень его универсальности. Такие типы напоминают константы, "зашитые" в тексте программы.

В некоторых ситуациях имеет смысл разрешить настройку типа переменной pointee_. Например, это стоит сделать при работе с нестандартными модификаторами указателя. Во времена 16-битовых процессоров Intel 80x86 указатели могли иметь спецификаторы _near, _far и _huge. Такие модификаторы могут использоваться и в других моделях сегментированной памяти.

Вторая ситуация, в которой следует настраивать тип, возникает, когда программист хочет применить наследование интеллектуальных указателей. Что если у нас есть класс LegacySmartPtr<T>, реализованный кем-то еще, и нам нужно создать производный от него класс? Как это сделать? Это ответственное решение. Лучше скрыть базовый класс внутри своего собственного интеллектуального указателя. Это вполне возможно, поскольку внутренний интеллектуальный указатель имеет ту же синтаксическую структуру. С точки зрения внешнего интеллектуального указателя переменная pointee_ имеет тип LegacySmartPtr<T>, а не T*.

Наследование интеллектуальных указателей обладает интересными приложениями, в основном благодаря оператору ->. Когда оператор -> применяется к типу, который отличается от встроенного, компилятор ведет себя необычно. После применения оператора ->, определенного пользователем, к данному типу он вновь применяет оператор -> к полученному результату и продолжает рекурсивно повторять этот процесс, пока не обнаружит указатель на встроенный тип и только тогда предоставит доступ к члену класса. Следовательно, оператор ->, определенный в интеллектуальном указателе, не обязан возвращать указатель. Он может возвращать объект, который в свою очередь реализует оператор ->, не изменяя синтаксис его применения.

Это приводит к интересной идиоме: вызовам пре- и постфункций (Stroustrup, 2000). Если оператор -> возвращает объект типа PointerType по значению, последовательность его выполнения такова.

1. Вызывается конструктор класса PointerType.
2. Вызывается функция PointerType::operator->; вероятно, возвращается указатель на объект типа PointeeType.
3. Доступ к члену класса PointeeType — вероятно, вызов функции.
4. Вызывается деструктор класса PointerType.

Короче говоря, у нас есть превосходный способ блокировки вызовов функций. Эта идиома широко применяется в многопоточных средах для блокировки доступа к ресурсам. Конструктор класса PointerType может захватывать ресурсы, мы их можем использовать в своих целях, а в конце работы деструктор класса PointerType освобождает их.

На этом обобщения не заканчиваются. Синтаксически-ориентированная часть "указателя" выглядит довольно бледно по сравнению с мощными средствами управления ресурсами, которыми обладают интеллектуальные указатели. Следовательно, иногда интеллектуальные указатели могут вести себя подобно обычным. Объект, для которого не определены операторы -> и *, не соответствует определению интеллекту-

ального указателя. Однако существуют объекты, которые можно считать интеллектуальными указателями, даже если они таковыми формально не являются.

Вернемся в реальный мир прикладного программирования и приложений. Многие операционные системы реализуют *дескрипторы* (handles) в виде методов доступа к определенным внутренним ресурсам, таким как окна, мьютексы или устройства. Дескрипторы являются преднамеренно замаскированными указателями. Одно из их предназначений — предотвращать непосредственный доступ пользователей к критически важным ресурсам операционной системы. В большинстве случаев дескрипторы представляют собой целые числа, указывающие на скрытую таблицу указателей. Эта таблица обеспечивает дополнительный уровень защиты внутренней системы от прикладных программистов. Хотя для дескрипторов не предусмотрен оператор ->, по семантике и способу работы они похожи на указатели.

Для дескрипторов нет смысла предусматривать оператор -> или *. Однако они предоставляют те же средства управления ресурсами, что и интеллектуальные указатели.

Рассмотрим три типа интеллектуальных указателей.

1. *Тип хранения* (storage type). Это — тип переменной `pointee_`. По умолчанию в обычных интеллектуальных указателях этот тип относится к обычным указателям.
2. *Тип указателя* (pointer type). Это тип объекта, возвращаемого оператором ->. Он может отличаться от типа хранения, если хранится объект-заместитель (proxy object), а не сам указатель. (Позднее в этой главе мы рассмотрим пример объекта-заместителя.)
3. *Ссылочный тип* (reference type). Это тип, возвращаемый оператором *.

В заключение заметим, что интеллектуальные указатели могут и должны обобщать все три перечисленных типа. Для этого класс `SmartPtr` абстрагирует их в стратегии `Storage`. Конкретизация класса `SmartPtr` не обязана реализовывать их все одновременно. Следовательно, иногда (например, в дескрипторах), стратегия может не определять один из операторов -> и *, либо игнорировать их обоих.

7.4. Функции-члены интеллектуальных указателей

Многие существующие реализации интеллектуальных указателей допускают выполнение операций с помощью функций-членов, например, `Get` для доступа к объекту, `Set` — для его изменения и `Release` — для отказа от владения. Вполне естественно инкапсулировать эти функциональные возможности в классе `SmartPtr`.

Однако опыт показывает, что функции-члены не очень удобны для реализации интеллектуальных указателей, поскольку взаимодействие между этими функциями крайне запутанно.

Допустим, что у нас есть класс `Printer` с функциями-членами `Acquire` и `Release`. С помощью функции-члена `Acquire` мы получаем права владения принтером, так что никакое другое приложение ничего напечатать не сможет, а с помощью функции-члена `Release` мы освобождаем принтер. Используя интеллектуальный указатель на объект класса `Printer`, мы обнаруживаем его странную синтаксическую схожесть с конструкциями, имеющими совершенно иную семантику.

```
SmartPtr<Printer> spRes = ...;
spRes->Acquire(); // Вступаем по владение принтером
... Печатаем документ ...
```

```
spRes->Release(); // Освобождаем принтер
spRes.Release();  // Освобождаем указатель на принтер
```

Пользователь класса `SmartPtr` теперь имеет доступ к двум совершенно разным множествам функций: функциям-членам объекта, на который ссылается интеллектуальный указатель, и функциям-членам самого интеллектуального указателя. Выбор этих функций зависит от используемого оператора: точки или стрелки.

Программисты на языке C++ вынуждены внимательно следить за тем, какой из операторов должен применяться в той или иной ситуации. Программисты на языке Pascal, изучающие язык C++, могут даже почувствовать отвращение к необходимости улавливать различия между операциями & и &&. Однако программисты на языке C++ не имеют права закрывать на это глаза. Они должны выработать в себе привычку легко обнаруживать такие синтаксические различия.

Привыкнуть к функциям-членам интеллектуальных указателей нелегко. Простые указатели не имеют функций-членов, поэтому операторы `.` и `->` к ним не применяются. В этих ситуациях большую помощь оказывает компилятор: если программист поставил после обычного указателя точку, выдается сообщение об ошибке. Теперь легко себе представить, как разочарованы даже опытные программисты на языке C++ тем фактом, что компилятор допускает использование выражений `sp.Release()` и `sp->Release()`, имеющих совершенно разный смысл. Избежать этого легко: интеллектуальный указатель не должен иметь функций-членов. Класс `SmartPtr` использует вместо них дружественные функции.

Перегруженные функции также могут привести к недоразумениям, но ситуация здесь совершенно иная. В языке C++ перегруженные функции применяются довольно широко. Перегрузка является важной частью языка и часто применяется в библиотеках и приложениях. Это значит, что программисты на языке C++ привыкли дифференцировать разные синтаксические формы вызова, например, `Release(*sp)` и `Release(sp)`.

Функциями-членами класса `SmartPtr` могут быть лишь конструкторы, деструктор, оператор =, оператор -> и унарный оператор *. Остальные операции, выполняемые классом `SmartPtr`, реализуются с помощью внешних функций.

Для простоты класс `SmartPtr` не использует именованные функции-члены. Единственными функциями, имеющими доступ к объекту, на который ссылается интеллектуальный указатель, являются `GetImpl`, `GetImplRef`, `Reset` и `Release`, определенные в пространстве имен.

```
template <class T> T* GetImpl(SmartPtr<T>& sp);
template <class T> T*& GetImplRef(SmartPtr<T>& sp);
template <class T> void Reset(SmartPtr<T>& sp, T* source);
template <class T> void Release(SmartPtr<T>& sp, T*& destination);
```

- Функция `GetImpl` возвращает указатель на объект, хранящийся в классе `SmartPtr`.
- Функция `GetImplRef` возвращает *ссылку* на указатель, хранящийся в классе `SmartPtr`. Она позволяет изменять этот указатель, поэтому требует особой осторожности.
- Функция `Reset` возвращает указатель на другое значение, освобождая предыдущее.
- Функция `Release` освобождает интеллектуальный указатель, возлагая на пользователя ответственность за управление объектом, на который он ссылался.

Реальные объявления этих функций в библиотеке Loki немного сложнее. В них не предполагается, что указатель, хранящийся в классе `SmartPtr`, имеет тип `T*`. Как указано в разделе 7.3, тип указателя определяется стратегией `Storage`.

7.5. Стратегии владения

Владение объектом является смыслом существования интеллектуальных указателей. Они сами выполняют уничтожение объектов, на которые ссылаются. В то же время пользователь может влиять на продолжительность жизни объекта, вызывая вспомогательные функции.

Для реализации прав владения интеллектуальный указатель должен внимательно следить за объектом, особенно во время копирования, присваивания и уничтожения. Это приводит к дополнительным затратам памяти и времени. Приложение должно определять стратегию, которая одновременно была бы удобной и эффективной.

В следующих подразделах обсуждаются наиболее распространенные стратегии владения и их реализации в классе `SmartPtr`.

7.5.1. Глубокое копирование

При копировании интеллектуального указателя простейшая стратегия заключается в копировании объекта, на который он ссылается. При этом на каждый объект будет ссылаться только один указатель. Следовательно, деструктор интеллектуального указателя может безопасно уничтожить такой объект. На рис. 7.1 проиллюстрирована схема распределения памяти для интеллектуальных указателей при глубоком копировании.

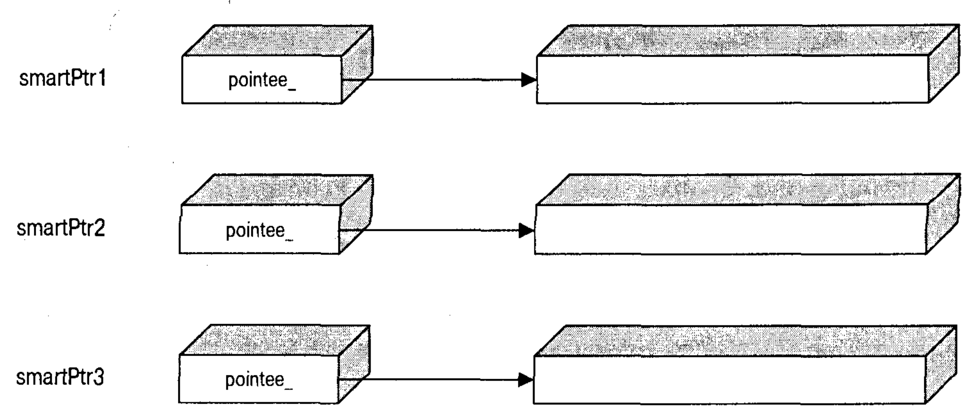


Рис. 7.1. Схема распределения памяти для интеллектуальных указателей при глубоком копировании

На первый взгляд стратегия глубокого копирования выглядит бессмысленной. Кажется, что в этом случае интеллектуальные указатели нисколько не расширяют обычную семантику значений языка C++. Зачем же их применять, если объект, на который они ссылаются, можно просто передавать по значению?

Ответ заключается в *поддержке полиморфизма*. Интеллектуальные указатели представляют собой средство для безопасной передачи полиморфных объектов. Интеллектуальный указатель на базовый класс может ссылаться и на производные классы. При копировании интеллектуального указателя требуется копировать и его полиморфное поведение. При этом ни его поведение, ни состояние точно не известны.

Поскольку глубокое копирование чаще всего применяется для полиморфных объектов, приведенная ниже наивная реализация оказывается неверной.

```
template <class T>
class SmartPtr
```

```
{
public:
   SmartPtr(const SmartPtr& other)
   : pointee_(new T(*other.pointee_))
   {
   }
   ...
};
```

Допустим, что мы копируем объект типа `SmartPtr<Widget>`. Если интеллектуальный указатель `other` ссылается на экземпляр класса `ExtendedWidget`, производного от класса `Widget`, конструктор копирования скопирует только ту часть объекта `ExtendedWidget`, которая унаследована им от класса `Widget`. Это явление известно как *срезка* (slicing) — копируется только "срез" класса `Widget`, содержащийся в объекте более широкого класса `ExtendedWidget`. Срезка чаще всего совершенно нежелательна. Очень жаль, что в языке C++ она возникает так легко — простой вызов по значению обрезает объекты без всякого предупреждения.

В главе 8 обсуждается глубокое клонирование. Классическим способом получения полиморфного клона иерархии является определение виртуальной функции `Clone` и ее реализация, как показано ниже.

```
class AbstractBase
{
   ...
   virtual Base* Clone() = 0;
};

class Concrete : public AbstractBase
{
   ...
   virtual Base* Clone()
   {
      return new Concrete(*this);
   }
};
```

Реализация функции `Clone` должна быть одинаковой во всех производных классах. Несмотря на такую повторяющуюся структуру, автоматического способа определения функции-члена `Clone` (кроме макросов) не существует.

Обобщенный интеллектуальный указатель не знает точно имя функции клонирования: может быть, `clone`, а может — `MakeCopy`. Следовательно, наиболее гибким подходом является параметризация класса `SmartPtr` с помощью соответствующей стратегии адресации клонирования.

7.5.2. Копирование при записи

Копирование при записи (copy on write — COW) — это способ оптимизации, позволяющий избежать необязательного копирования объекта. Идея этого метода заключается в том, чтобы клонировать объект лишь при первой попытке его модификации, а до тех пор на него может ссылаться несколько указателей.

Однако интеллектуальные указатели не очень подходят для реализации стратегии COW, поскольку они не умеют различать вызовы константных и неконстантных функций-членов объекта, на который они ссылаются. Рассмотрим следующий пример.

```
template <class T>
class SmartPtr
```

```
{
public:
   T* operator->() { return pointee_; }
   ...
};

class Foo
{
public:
   void ConstFun() const;
   void NonConstFun();
};

...
SmartPtr<Foo> sp;
sp->ConstFun();    // Вызывает оператор ->,
                   // а затем — функцию ConstFun
sp->NonConstFun()  // Вызывает оператор ->,
                   // а затем — функцию NonConstFun
```

Для обеих функций вызывается один и тот же оператор ->. Следовательно, интеллектуальный указатель не может решить, применять стратегию COW или нет. Вызовы функций объекта иногда выходят за рамки возможностей интеллектуальных указателей (в разделе 7.11 поясняется, как ключевое слово `const` влияет на интеллектуальные указатели и объекты, на которые они ссылаются).

В заключение отметим, что стратегия COW наиболее эффективна при оптимизации полноценных классов. Интеллектуальные указатели находятся на слишком низком уровне, чтобы применять к ним копирование при записи. Разумеется, они, в свою очередь, могут представлять собой хорошие строительные блоки для реализации стратегии COW в каком-нибудь другом классе.

7.5.3. Подсчет ссылок

Подсчет ссылок (reference counting) — наиболее распространенная стратегия владения объектом, используемая интеллектуальными указателями. В рамках этой стратегии производится подсчет интеллектуальных указателей, ссылающихся на один и тот же объект. Когда их количество становится равным нулю, объект уничтожается. Эта стратегия работает очень хорошо, если не нарушаются несколько правил — например, на один и тот же объект не должны ссылаться и обычный, и интеллектуальный указатели.

Подсчет ссылок должен распространяться только на интеллектуальные указатели. Это приводит к структуре, изображенной на рис. 7.2. Каждый интеллектуальный указатель, кроме самого объекта, хранит указатель на счетчик ссылок (переменную `pRefCount_` на рис. 7.2). Обычно это приводит к удвоению размера интеллектуального указателя, что может оказаться нежелательным.

Существует еще один вопрос, связанный с расходом ресурсов. Интеллектуальные указатели с подсчетом ссылок должны хранить в динамической памяти счетчик ссылок. Проблема заключается в том, что во многих реализациях механизм распределения динамической памяти, предусмотренный по умолчанию, работает довольно медленно и неэффективно использует память при выделении ее для небольших объектов (глава 4). (Очевидно, счетчик ссылок, обычно занимающий 4 байт, следует считать небольшим объектом.) Потеря скорости происходит из-за медленного алгоритма поиска доступных участков памяти (chunks), а затраты памяти вызваны необходимостью хранить информацию о каждом таком участке.

Относительный перерасход памяти может частично компенсироваться совместным хранением указателя и счетчика ссылок, как показано на рис. 7.3. Структура, изображенная на рис. 7.3, позволяет сократить размер интеллектуального указателя до размера обычного указателя, но за счет скорости доступа: объект, на который ссылается интеллектуальный указатель создает дополнительный уровень косвенной адресации. Это довольно значительный недостаток, поскольку обычно интеллектуальные указатели используются несколько раз, а создаются и уничтожаются — лишь однажды.

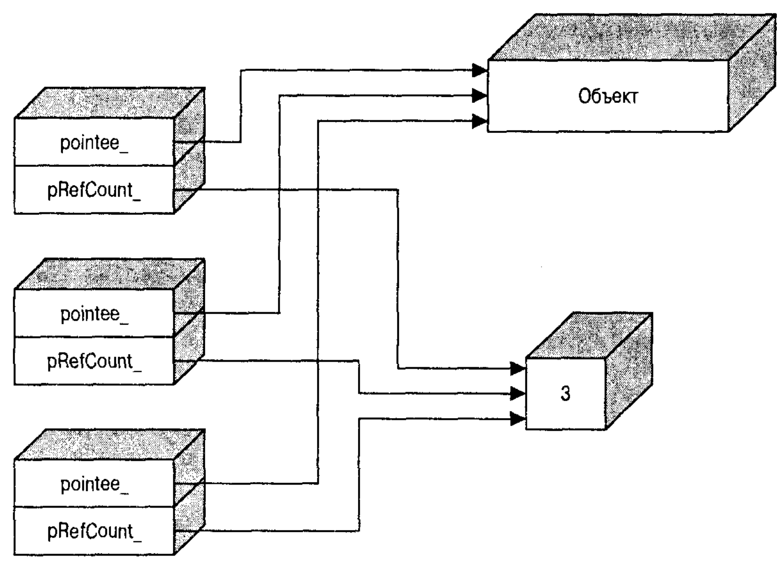


Рис. 7.2. Три интеллектуальных указателя, ссылающихся на один и тот же объект

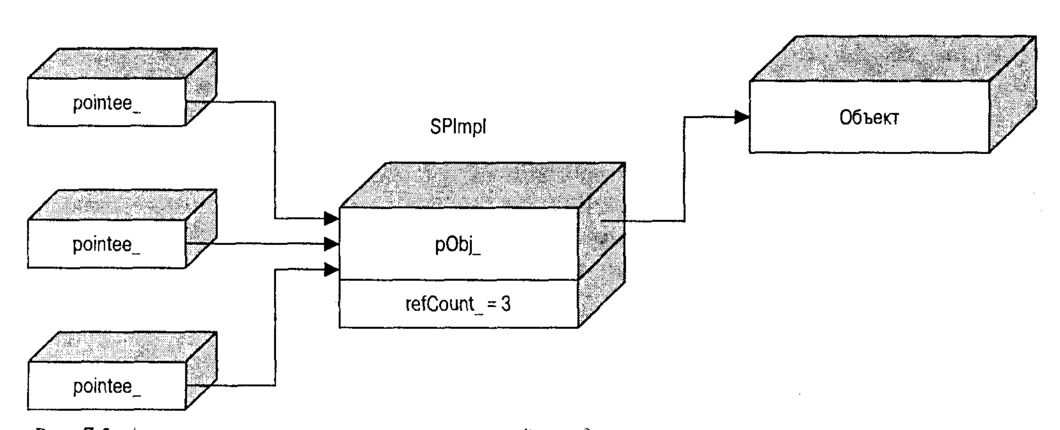


Рис. 7.3. Альтернативная структура указателей с подсчетом ссылок

Эффективнее всего хранить счетчик ссылок в самом объекте, на который ссылается интеллектуальный указатель, как показано на рис. 7.4. Таким образом, объект класса `SmartPtr` будет иметь размер обычного указателя, и дополнительных затрат памяти не будет совсем. Этот прием называется *внедренным подсчетом ссылок* (intrusive reference counting), поскольку счетчик ссылок "внедряется" в объект, хотя семантиче-

ски он относится к интеллектуальному указателю. Его название напоминает также об ахиллесовой пяте этого приема: для реализации такой стратегии нужно модифицировать класс, которому принадлежит объект.

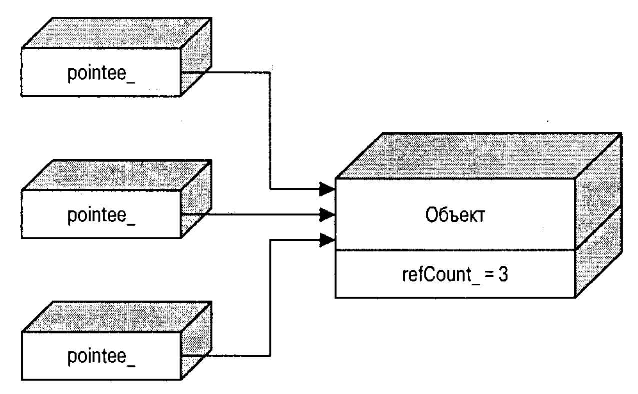


Рис. 7.4. Внедренный подсчет ссылок

Обобщенный интеллектуальный указатель должен при первой возможности использовать внедренный подсчет ссылок, а обычную схему подсчета ссылок следует рассматривать в качестве приемлемой альтернативы. Для реализации обычной схемы подсчета ссылок очень удобно применять механизм распределения памяти для небольших объектов, описанный в главе 4. Он позволяет снижать дополнительные затраты, вызванные необходимостью хранить счетчик ссылок.

7.5.4. Связывание ссылок

На самом деле нет никакой необходимости подсчитывать интеллектуальные указатели, ссылающиеся на один и тот же объект. Нужно лишь определять, когда этот счетчик станет равным нулю. Это приводит к идее "списка владельцев" (ownership list), показанного на рис. 7.5,[1] и стратегии связывания ссылок.

Все объекты класса `SmartPtr`, ссылающиеся на заданный объект, заносятся в дважды связанный список. Новый объект класса `SmartPtr`, создаваемый на основе существующего, добавляется в список, а деструктор этого класса удаляет уничтоженные объекты из списка. Когда список становится пустым, объект удаляется.

Структура дважды связанного списка идеально подходит для отслеживания ссылок. Использовать односвязный список невозможно, поскольку удаление элемента из такого списка занимает линейное время. Вектор применить нельзя, так как объекты класса `SmartPtr` не занимают смежные участки памяти (и на удаление элементов из вектора также тратится линейное время). Нам нужна структура, в которой удаление и вставка элементов, а также проверка заполненности занимали бы одинаковое время. Единственной структурой, удовлетворяющей этим требованиям, является дважды связанный список.

В реализации стратегии связывания ссылок в каждом объекте класса `SmartPtr` хранятся два дополнительных указателя — на следующий и на предыдущий элементы.

---

[1] Механизм связывания ссылок описан Ристо Ланкиненом (Risto Lankinen) в форуме Usenet в ноябре 1995 года.

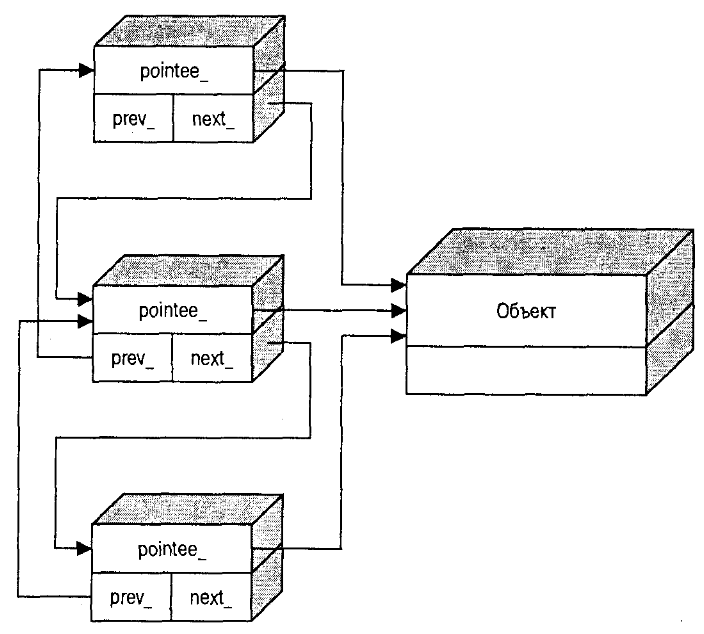


Рис. 7.5. Связывание ссылок

Преимущество связывания ссылок перед их подсчетом проявляется в том, что при первом способе не требуется дополнительная память. Это делает стратегию более надежной. Создание интеллектуального указателя со связыванием ссылок всегда работает безотказно. Недостатком этой стратегии является тот факт, что при связывании необходимо больше памяти для регистрации (три указателя вместо одного плюс одно целое число). Кроме того, подсчет ссылок выполняется немного быстрее — при копировании интеллектуального указателя необходима только одна косвенная адресация и одна операция инкрементации. Управление списком также немного сложнее. В заключение отметим, что связывание ссылок следует применять только при недостатке динамической памяти. В противном случае предпочтительнее использовать подсчет ссылок.

Завершая обсуждение стратегий управления ссылками, обратим внимание на их существенный недостаток. Управление ссылками — с помощью подсчета или связывания — приводит к утечке ресурсов, так называемой *циклической ссылке* (cyclic reference). Представьте себе, что объект `A` содержит интеллектуальный указатель на объект `B`, и, наоборот, — в объекте `B` хранится интеллектуальный указатель на объект `A`. Эти два объекта образуют циклическую ссылку. Даже если вы не используете эти объекты, они сами друг друга используют. Стратегии управления ссылками не способны распознавать такие ситуации, и эти два объекта останутся в динамической памяти навсегда. Циклы могут распространяться на несколько объектов, создавая между ними связи, которые очень трудно отследить.

Несмотря на это, управление ссылками представляет собой надежную и быструю стратегию владения объектами. Если применять его с необходимыми предосторожностями, оно значительно облегчает разработку приложений.

7.5.5. Разрушающее копирование

Разрушающее копирование (destructive copy) — это именно то, о чем вы подумали: во время копирования оригинал уничтожается. Разрушающее копирование уничтожа-

ет оригинальный указатель, передавая объект, на который он ссылался, другому интеллектуальному указателю. В шаблонном классе std::auto_ptr применяется именно этот вид копирования.

Опасность этой стратегии очевидна. Неправильное разрушающее копирование может уничтожить данные, программу и вашу репутацию программиста.

Эту стратегию следует применять только в том случае, когда в каждый момент времени на данный объект ссылается только один интеллектуальный указатель. Во время копирования или присваивания одного интеллектуального указателя другому "в живых" остается только результирующий указатель, а оригинал обнуляется. Ниже показаны конструктор копирования и оператор присваивания простого класса SmartPtr, демонстрирующие разрушающее копирование.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(SmartPtr& src)
   {
      pointee_ = src.pointee_;
      src.pointee_ = 0;
   }
   SmartPtr& operator=(SmartPtr& src)
   {
      if (this != &src)
      {
         delete pointee_;
         pointee_ = stc.pointee_;
         src.pointee_ = 0;
      }
      return *this;
   }
   ...
};
```

По правилам языка C++ в правой части конструктора копирования и оператора присваивания должна стоять ссылка на константный объект. Классы, реализующие разрушающее копирование, нарушают это правило по очевидным причинам. Поскольку правила языка основаны на резонных соображениях, от их нарушения нельзя ожидать ничего хорошего. Действительно, рассмотрим следующий пример.

```
void Display(SmartPtr<Something> sp);
...
SmartPtr<Something> sp(new Something);
Display(sp); // Уничтожает объект sp
```

Несмотря на то что функция Display не представляет опасности для своего аргумента (принимая его по значению), она действует как водоворот: все интеллектуальные указатели, попавшие в нее, тонут безвозвратно. После вызова Display(sp) указатель sp хранит нулевой адрес.

Поскольку интеллектуальные указатели, реализующие стратегию разрушающего копирования, не поддерживают семантику значений, их нельзя хранить в контейнерах и вообще с ними нужно обращаться почти так же осторожно, как и с обычными указателями.

Возможность хранить интеллектуальные указатели в контейнере чрезвычайно важна, поскольку контейнеры, состоящие из обычных указателей, очень усложняют ручное управление владением. Однако интеллектуальные указатели, реализующие стратегию разрушающего копирования, для этого не подходят.

С другой стороны, интеллектуальные указатели, реализующие стратегию разрушающего копирования, имеют ряд преимуществ.

- Они почти не тратят дополнительной памяти.
- Они удобны для передачи владения другому указателю. В этом случае используется "эффект водоворота", описанный выше.
- Они удобны в качестве возвращаемого функцией значения. Если в реализации интеллектуального указателя используется определенный трюк[2], можно возвращать из функций интеллектуальные указатели с разрушающим копированием. Таким образом, можно быть уверенным, что если в вызывающем модуле возвращаемое значение не используется, оно уничтожается.
- Они превосходны в качестве переменных стека в функциях, возвращающих значения несколькими способами. Предпринимать меры для уничтожения объекта, на который ссылается интеллектуальный указатель, необязательно — он сам это сделает.

Стратегия разрушающего копирования используется в стандартном классе `std::auto_ptr`. Это создает дополнительное преимущество.

- Интеллектуальные указатели, обладающие семантикой разрушающего копирования, являются единственным стандартом. Это значит, что многие программисты рано или поздно станут их использовать.

По этим причинам реализация класса `SmartrPtr` должна предусматривать дополнительную поддержку семантики разрушающего копирования.

Интеллектуальные указатели используют разные стратегии владения, каждая из которых имеет свои достоинства и недостатки. К наиболее важным приемам относятся глубокое копирование, подсчет ссылок, связывание ссылок и разрушающее копирование. Класс `SmartPtr` реализует все эти способы в виде стратегии `Ownership`, позволяя своим пользователям выбирать из них наиболее подходящий. По умолчанию предлагается стратегия подсчета ссылок.

7.6. Оператор взятия адреса

Стремясь сделать интеллектуальные указатели максимально похожими на их обычные прототипы, разработчики натолкнулись на незаметный перегружаемый унарный оператор &, или *оператор взятия адреса* (address-of operator)[3].

Программист, реализующий интеллектуальные указатели, может перегрузить этот оператор следующим образом.

```
template <class T>
class SmartPtr
{
public:
   T** operator&()
   {
      return &pointee_;
```

[2] Изобретенный Грегом Колвином (Greg Colvin) и Биллом Гиббонсом (Bill Gibbons) для стандартного интеллектуального указателя `std::auto_ptr`.

[3] *Унарный* оператор & следует отличать от бинарного оператора &, представляющего собой побитовый оператор AND.

```
    }
    ...
};
```

Помимо всего прочего, если интеллектуальный указатель должен имитировать обычный указатель, то его адрес можно заменять адресом обычного указателя. В этом случае становится возможной следующая перегрузка оператора.

```
void Fun(Widget** pWidget);
...
SmartPtr<Widget> spWidget(...);
Fun(&spWidget); // Вызывает оператор * и получает указатель
                // на указатель на объект класса Widget
```

На первый взгляд было бы замечательно иметь полную совместимость интеллектуальных и обычных указателей, однако перегрузка унарного оператора & относится к тем остроумным трюкам, которые приносят больше вреда, чем пользы.

Есть две причины, по которым перегрузка унарного оператора & опасна. Первая заключается в том, что явное определение адреса объекта, на который ссылается интеллектуальный указатель, отнимает возможности автоматического управления владением. Если пользователь имеет свободный доступ к адресу указателя, любая вспомогательная структура, содержащаяся в интеллектуальном указателе, например, счетчик ссылок, становится неработоспособной. Если пользователь непосредственно оперирует адресом обычного указателя, интеллектуальный указатель становится полностью неуправляемым.

Вторая причина более прагматична. Дело в том, что перегрузка унарного оператора & делает невозможным использование интеллектуального указателя в сочетании с контейнерами из стандартной библиотеки шаблонов STL. Фактически перегрузка унарного оператора & не позволяет применять обобщенное программирование, поскольку адрес любого объекта — слишком важное свойство, чтобы так просто с ним обращаться. В большинстве обобщенных кодов предполагается, что применение оператора & к объекту типа T возвращает объект типа T*. Как видим, оператор взятия адреса представляет собой фундаментальное понятие обобщенного программирования. Если им пренебречь, обобщенный код будет вести себя странно как на этапе компиляции, так и (что намного хуже) во время выполнения программы.

Таким образом, перегружать унарный оператор & для интеллектуальных указателей и вообще для любых объектов не рекомендуется, поэтому в классе SmartPtr унарный оператор & не перегружается.

7.7. Неявное приведение к типам обычных указателей

Рассмотрим следующий код.

```
void Fun(Something* p)
...
SmartPtr(Something> sp(new Something);
Fun(sp); // Правильно или нет?
```

Скомпилируется этот код или нет? По принципу "максимальной совместимости" правильный ответ — "да".

С технической точки зрения сделать приведенный выше код компилируемым очень легко. Для этого достаточно ввести преобразование, определенное пользователем.

```
template <class T>
class SmartPtr
{
```

```
public:
   operator T*() // Приведение к типу T*,
                 // определенное пользователем
   {
      return pointee_;
   }
   ...
};
```

Однако это еще не все.

Преобразования типов, определенные пользователем, имеют весьма интересную историю. В 1980-х годах, когда они впервые появились в языке C++, большинство программистов считали их огромным достижением. Они позволяли разрабатывать единообразные системы типов, повышать выразительность семантики и определять новые типы, отличающиеся от встроенных. Однако впоследствии оказалось, что преобразования типов, определенные пользователем, неудобны и потенциально опасны. Они могут стать опасными особенно при обработке внутренних данных (Meyers, 1998a, Item 29). Именно это происходит с типом `T*` в приведенном выше коде. Вот почему следует хорошенько подумать, прежде чем допускать автоматическое преобразование интеллектуальных указателей в вашей программе.

Одна из потенциальных опасностей происходит от неконтролируемого доступа пользователя к обычному указателю, который скрыт внутри интеллектуального. Передача обычного указателя во внешнюю среду нарушает внутреннюю работу интеллектуального указателя. Вырвавшись из своей оболочки, обычный указатель может легко стать угрозой для нормального функционирования программы, как это было до появления интеллектуальных указателей.

Другая опасность исходит от непредвиденного выполнения преобразований, определенных пользователем, даже когда они совершенно не нужны. Рассмотрим следующий пример.

```
SmartPtr<Something> sp;
...
// Грубая семантическая ошибка.
// Однако компилятор ее не распознает
delete sp;
```

Компилятор сопоставляет оператор `delete` с преобразованием к типу `T*`, определенным пользователем. Во время выполнения программы вызывается оператор `T*`, и к результату его работы применяется оператор `delete`. Очевидно, это совершенно не то, что требуется от интеллектуального указателя, поскольку предполагается, что он сам управляет правами владения. Лишний и непреднамеренный вызов оператора `delete` нарушает всю тщательно отлаженную систему управления правами владения, скрытую внутри интеллектуального указателя.

Есть несколько способов преодотвратить компиляцию вызова оператора `delete`. Некоторые из них очень остроумны (Meyers, 1996). Эффективнее и удобнее всего сделать вызов оператора `delete` внутренне *неоднозначным* (ambiguous). Этого можно достичь, предусмотрев *два* приведения к типам, реагирующих на вызов оператора `delete`. Один из типов — это сам тип `T*`, а вторым может быть тип `void*`.

```
template <class T>
class SmartPtr
{
public:
   operator T*() // Приведение к типу T*,
                 // определенное пользователем
```

```
    {
       return poiintee_;
    }
    operator void*() // Добавляется приведение к типу void*
    {
       return pointee_;
    }
    ...
};
```

Применение оператора delete к интеллектуальному указателю неоднозначно. Компилятор не сможет решить, какое из преобразований следует выполнить. Именно этого мы и добивались.

Не забывайте, что блокировка оператора delete — это только часть проблемы. Остается решить, применять ли автоматическое преобразование интеллектуального указателя в простой указатель. Разрешить — слишком опасно, запретить — очень легко. Окончательная реализация класса SmartPtr предоставляет программисту право выбора.

Однако запрет неявного преобразования не означает полного исключения доступа к обычному указателю. Иногда такой доступ просто необходим. Следовательно, все интеллектуальные указатели должны обеспечивать *явный* доступ к скрытому внутри них обычному указателю с помощью вызова функции.

```
void Fun(Something* p)
...
SmartPtr<Something> sp;
Fun(GetImpl(sp)); // Явное преобразование всегда разрешено
```

Вопрос заключается не в том, можете ли вы получить доступ к обычному указателю, а в том, насколько это легко. Может показаться, что разница между явным и неявным преобразованиями невелика, однако она очень важна. Неявное преобразование происходит независимо от программиста, он может даже ничего не знать о нем. Явное преобразование, как, например, вызов функции GetImpl, выполняется осознанно, управляется программистом и не скрывается от пользователей программы.

Неявное преобразование интеллектуального указателя в простой указатель желательно, но иногда опасно. Класс SmartPtr предоставляет программисту право самому решать, допускать его или нет. По умолчанию предлагается безопасный вариант — неявное преобразование запрещается. Явный доступ к обычному указателю всегда открыт через функцию GetImpl.

7.8. Равенство и неравенство

Любой трюк в языке C++, например, описанный выше (внутренняя неоднозначность), создает новый контекст, который в свою очередь может иметь неожиданные последствия.

Рассмотрим проверку интеллектуальных указателей на равенство и неравенство. Интеллектуальный указатель должен поддерживать такой же синтаксис проверки, что и обычные указатели. Программист ожидает, что приведенные ниже проверки будут скомпилированы и выполнены.

```
SmartPtr<Something> sp1, sp2;
Something* p;
...
if (sp1) // Проверка 1: прямая проверка ненулевого указателя
```

```
    ...
if (!sp1) // Проверка 2: прямая проверка нулевого указателя
    ...
if (sp1 == 0) // Проверка 3: явная проверка нулевого указателя
    ...
if (sp1 == sp2) // Проверка 4: сравнение двух интеллектуальных указателей
    ...
if (sp1 == p) // Проверка 5: сравнение с обычным указателем
    ...
```

В этом фрагменте проиллюстрированы не все возможные проверки. Реализовав проверку равенства, мы сможем легко проверять и неравенства.

К сожалению, между проблемой проверки равенства и проблемой предотвращения компиляции оператора `delete` существует скрытая связь. Если в программе определено только одно преобразование, большинство проверок (за исключением проверки 4) проходит успешно, и программа выполняется в соответствии с ожиданиями программиста. Единственный недостаток — возможность непредвиденного применения оператора `delete` к интеллектуальному указателю. Если в программе определены два преобразования (внутренняя неоднозначность), неверные вызовы оператора `delete` распознаются, но ни одна из указанных выше проверок больше не компилируется — они также становятся неоднозначными.

Дополнительное преобразование интеллектуального указателя в булевскую переменную является полезным, но небезопасным. Рассмотрим следующее определение интеллектуального указателя.

```
template <class T>
class SmartPtr
{
public:
   operator bool() const
   {
      return pointee_ != 0;
   }
   ...
};
```

Четыре указанные выше проверки успешно компилируются, но при этом выполняются следующие бессмысленные операции.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2; // Яблоко и апельсин — не одно и то же
if (sp1 == sp2)       // Оба указателя преобразовываются
                      // в булевскую переменную,
                      // а результаты сравниваются
    ...
if (sp1 != sp2)       // То же самое
    ...
bool b = sp1;         // Допустимая операция
if (sp1 * 5 == 200)   // Ой! Интеллектуальный указатель
                      // ведет себя как целое число!
    ...
```

Как видим, мы можем получить либо все, либо ничего. Добавив преобразование интеллектуального указателя в булевскую переменную, мы допустили ситуации, когда объект класса `SmartPtr` ведет себя неправильно. Таким образом, определение оператора `bool` для интеллектуального указателя оказалось неразумным решением.

Истинное, всеобъемлющее и надежное решение этой дилеммы заключается в полной перегрузке всех операторов по отдельности. При таком подходе все операции, имеющие смысл для обычных указателей, окажутся применимыми и для интеллектуальных указателей без каких-либо побочных эффектов. Вот как можно реализовать эту идею.

```
template <class T>
class SmartPtr
{
public:
   bool operator!() const // Допускает оператор "if (!sp) ..."
   {
      return pointee_ = 0;
   }
   inline friend bool operator==(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   inline friend bool operator==(const T* lhs,
      const SmartPtr* rhs)
   {
      return lhs == rhs.pointee_;
   }
   inline friend bool operator!=(const SmartPtr& lhs,
      const T* rhs)
   {
      return lhs.pointee_ != rhs;
   }
   inline friend bool operator!=(const T* lhs,
     const SmartPtr& rhs)
   {
      return lhs != rhs.pointee_;
   }
   ...
};
```

Да, это неприятно, однако этот подход решает проблемы, касающиеся практически всех сравнений, включая сравнение с литеральным нулем. Операторы пересылки, содержащиеся в этом фрагменте кода, передают операторы, которые код пользователя применяет к интеллектуальным указателям, обычным указателям, скрытым внутри. Это самое реалистичное решение.

Однако проблема решена не полностью. Если в программе предусматривается автоматическое преобразование указателей, остается риск, связанный с неоднозначностью такой операции. Допустим, у нас есть класс `Base` и класс `Derived`, производный от него. Тогда в следующем коде будет содержаться неоднозначность.

```
SmartPtr<Base> sp;
Derived* p;
...
if (sp == p) {} // Ошибка! Существует неоднозначность между
                // проверкой '(Base*)sp == (Base*)p' и
                // функцией 'operator==(sp, (Base*)p)'
```

Действительно, разработка интеллектуальных указателей — занятие не для слабонервных.

Но и это еще не все. Кроме определения операторов `==` и `!=`, мы можем создать их *шаблонные* версии.

```
template <class T>
class SmartPtr
```

```
{
public:
   ...  как и раньше ...
   template <class U>
   inline friend bool operator==(const SmartPtr& lhs,
      const U* rhs)
   {
      return lhs.pointee_ == rhs;
   }
   template <class U>
   inline friend bool operator==(const U* lhs,
      const SmartPtr& rhs)
   {
      return lhs == rhs.pointee_;
   }
   ... аналогично определенный оператор != ...
};
```

Шаблонные операторы выполняют сравнения с любым типом указателей, устраняя неоднозначность.

Если все так хорошо, зачем сохранять обычные, не шаблонные операторы, работающие с типами объектов, на которые ссылаются указатели? Эти операторы никогда не пройдут процедуру сопоставления, потому что шаблоны сопоставляют любые типы указателей, в том числе и типы объектов, на которые ссылаются указатели.

Однако "никогда не говори *никогда*". При выполнении проверки `if (sp == 0)` компилятор пытается выполнить следующие сопоставления.

- *Шаблонные операторы* (templated operators). Они не проходят проверку, поскольку нуль не является типом указателей. Литеральный нуль можно неявно преобразовать в тип указателей, однако шаблонное сопоставление не предусматривает преобразований типов.
- *Нешаблонные операторы* (nontemplated operators). Исключив шаблонные операторы, компилятор переходит к проверке нешаблонных. Один из этих операторов оказывается подходящим после выполнения неявного преобразования литерального нуля в тип указателя. Если бы нешаблонных операторов не было, проверка завершилась бы сообщением об ошибке.

Итак, и шаблонные, и нешаблонные операторы *одинаково необходимы*.

Посмотрим теперь, что произойдет при попытке сравнения двух классов `SmartPtr`, конкретизированных разными типами.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2)
   ...
```

Компилятор споткнется на этом сравнении из-за неоднозначности: в каждой из этих двух конкретизаций определен оператор ==, и компилятор не знает, который из них выбрать. Мы можем избежать этой проблемы, определив "ликвидатор неоднозначностей".

```
template <class T>
class SmartPtr
{
public:
   // Ликвидатор неоднозначностей
   template <class U>
   bool operator==(const SmartPtr<U>& rhs) const
```

```
    {
        return pointee_ == rhs.pointee_;
    }
    // Аналогично для оператора !=
    ...
};
```

Новый оператор является членом класса, предназначенным исключительно для сравнения объектов классов SmartPtr<...>. Прелесть этого ликвидатора неоднозначностей заключается в том, что он сравнивает интеллектуальные указатели, как обычные указатели. Если сравнить два интеллектуальных указателя на объекты классов Apple и Orange, код будет, по существу, эквивалентным сравнению обычных указателей на классы Apple и Orange. Если это сравнение имеет смысл, код компилируется, в противном случае выдается сообщение об ошибке.

```
SmartPtr<Apple> sp1;
SmartPtr<Orange> sp2;
if (sp1 == sp2) // Семантически эквивалентно сравнению
                // sp1.pointee_ == sp2.pointee_
    ...
```

Остался один неприятный артефакт — непосредственная проверка if (sp). Вот это уже действительно интересно! Оператор if применяется только к выражениям арифметического типа или к указателям. Следовательно, чтобы скомпилировать оператор if (sp), нужно определить автоматическое преобразование интеллектуального указателя в арифметический тип или в обычный указатель.

Преобразовывать интеллектуальный указатель в арифметический тип не рекомендуется по тем же причинам, которые были указаны при преобразовании в булевский тип. Указатель — это не арифметический тип, и точка! Преобразование интеллектуального указателя в обычный имеет больше смысла, но тут возникают новые проблемы.

Если нужно преобразовать интеллектуальный указатель в тип, на который он ссылается (см. предыдущий раздел), у нас есть выбор: либо рисковать, допуская возможность непредвиденного вызова оператора delete, или воздержаться от проверки if (sp). Что выбрать: неудобство или опасность? Следует предпочесть безопасность, поэтому проверку if (sp) выполнять не следует. Вместо этого можно выбрать проверку if(sp != 0) или более вычурное выражение if(!!sp).

Если вы не хотите предусматривать преобразование интеллектуального указателя в тип, на который он ссылается, можно прибегнуть к интересному трюку, который позволит все же выполнить проверку if (sp). Внутри шаблонного класса SmartPtr нужно определить внутренний класс Tester и преобразование в тип Tester*, как показано ниже.

```
template <class T>
class SmartPtr
{
    class Tester
    {
        void operator delete(void*);
    }
public:
    operator Tester*() const
    {
        if (!pointee_) return 0;
        static Tester test;
        return &test;
    }
```

Теперь при компиляции оператора if (sp) в действие вступит оператор Tester*. Этот оператор возвращает нулевое значение тогда и только тогда, когда переменная pointee_ равна нулю. Класс Tester блокирует оператор delete, поэтому, если какой-нибудь код вызовет оператор delete sp, возникнет ошибка компиляции. Обратите внимание на то, что определение самого класса Tester находится в закрытом разделе класса SmartPtr, поэтому клиентский код ничего не знает о нем.

Класс SmartPtr решает проблемы, связанные с проверкой равенства и неравенства, следующим образом.

- Определяются операторы == и != двух видов (шаблонные или нешаблонные).
- Определяется оператор !.
- Если допускается автоматическое преобразование интеллектуального указателя в тип, на который он ссылается, то определяется дополнительное преобразование в тип void*, преднамеренно создающее неоднозначность при вызове оператора delete. В противном случае определяется внутренний класс Tester, объявляющий закрытый оператор delete и определяющий преобразование класса SmartPtr в тип Tester*, возвращающее нулевой указатель тогда и только тогда, когда объект, на который ссылается интеллектуальный указатель, является нулевым.

7.9. Отношения порядка

К операторам отношения порядка относятся операторы <, <=, > и >=. Все эти операторы можно свести к одному оператору <.

Допускать ли упорядоченность интеллектуальных указателей — вопрос интересный сам по себе. Он основан на двойственной природе указателей, которая часто смущает программистов. Указатели одновременно являются итераторами и моникерами (monikers). Будучи итераторами, указатели могут перемещаться по массиву объектов. Арифметика указателей, включая операции сравнения, поддерживает их итеративную природу. В то же время указатели являются моникерами — удобным способом быстрого доступа к объектам. Эту ипостась указателей обеспечивают операторы разыменования * и ->.

Двойственная природа указателей иногда может приводить в замешательство, особенно когда указатель нужно использовать только в одном качестве. Для работы с векторами может понадобиться как перемещение по массиву, так и разыменование, в то время как перемещение по связанному списку или манипуляция индивидуальными объектами используют только операцию разыменования.

Отношения порядка между указателями определены только для указателей, принадлежащих непрерывному участку памяти. Иными словами, сравнивать между собой можно лишь те указатели. которые ссылаются на элементы одного и того же массива.

Определение отношений порядка для интеллектуальных указателей порождает вопрос: могут ли интеллектуальные указатели ссылаться на объекты, хранящиеся в одном и том же массиве? На первый взгляд нет. Основное свойство интеллектуальных указателей заключается в том, что они управляют правами владения объектами, а объекты с разными правами владения не могут принадлежать одному и тому же массиву. Следовательно, было бы рискованно выполнять бессмысленные сравнения.

Если отношения порядка действительно необходимы, всегда можно применить явный доступ к обычному указателю, хранящемуся внутри интеллектуального. Здесь снова нужно искать наиболее безопасный и выразительный способ сравнения.

В предыдущем разделе мы пришли к выводу, что неявное преобразование интеллектуального указателя в обычный — вопрос выбора. Если пользователь класса

SmartPtr позволяет неявное преобразование, то приведенный ниже код будет успешно скомпилирован.

```
SmartPtr<Something> sp1, sp2;
if (sp1 < sp2) // Превращаем интеллектуальные указатели
               // sp1 и sp2 в обычные, а затем сравниваем их
...
```

Это значит, что блокировать отношения порядка следует явно. Для этого достаточно объявить их, но не определять. При попытке выполнить такую проверку во время редактирования связей возникнет ошибка.

```
template <class T>
class SmartPtr
{ ... };

template <class T, class U>
bool operator<(const SmartPtr<T>&, const U&); // Не определен
template <class T, class U>
bool operator<(const T&, const SmartPtr<U>&); // Не определен
```

Благоразумнее выразить остальные операторы через оператор <, оставив его неопределенным. Таким образом, если пользователь класса SmartPtr все же захочет установить между его объектами отношение порядка, ему останется лишь определить соответствующий оператор <.

```
// Ликвидатор неоднозначностей
template <class T, class U>
bool operator<(const SmartPtr<T>& lhs, const SmartPtr<U>& rhs)
{
   return lhs < GetImpl(rhs);
}
// Все остальные операторы
template <class T, class U>
bool operator>(SmartPtr<T>& lhs, const U& rhs)
{
   return rhs < lhs;
}
... аналогично для остальных операторов ...
```

Обратите внимание на ликвидатор неоднозначностей. Теперь, если какой-нибудь пользователь библиотеки полагает, что объекты класса SmartPtr<Widget> должны быть упорядочены, он может написать следующий код.

```
inline bool operator<(const SmartPtr<Widget>& lhs,
   const Widget* rhs)
{
   return GetImpl(lhs) < rhs;
}

inline bool operator<(const Widget* lhs,
  const SmartPtr<Widget>& rhs)
{
   return lhs < GetImpl(rhs);
}
```

Жаль, что пользователь должен определять два оператора, а не один, но это все же лучше, чем если бы он определял все восемь операторов.

На этом обсуждение отношения порядка между интеллектуальными указателями можно было бы закончить. Ничего интересного в этой проблеме обнаружить не удалось. Иногда очень полезно установить отношение порядка между произвольно раз-

мещенными в памяти объектами, а не только между объектами, хранящимися в одном и том же массиве. Например, иногда нужно хранить вспомогательную информацию о каждом объекте и быстро ее извлекать. Упорядоченный ассоциативный массив (map) адресов этих объектов — очень эффективное решение этой проблемы.

Стандарт языка C++ позволяет легко воплощать такие замыслы. Хотя сравнение указателей на произвольно размещенные объекты не определено, стандарт гарантирует, что оператор `std::less` даст осмысленные результаты, если применить его к двум указателям, имеющим одинаковый тип. Поскольку стандартные ассоциативные контейнеры используют оператор `std::less` в качестве отношения порядка, заданного по умолчанию, можно вполне безопасно пользоваться ассоциативными массивами, ключами которых являются указатели.

Класс `SmartPtr` также поддерживает эту идиому. Следовательно, класс `SmartPtr` осуществляет специализацию оператора `std::less`. Эта специализация заключается в простой переадресации вызова оператору `std::less` для обычных указателей.

```
namespace std
{
   template <class T>
   struct less<SmartPtr<T> >
      : public binary_function<SmartPtr<T>, SmartPtr<T>, bool>
   {
      bool operator() const SmartPtr<T>& lhs,
         const SmartPtr<T>& rhs) const
         {
            return less<T*>()(GetImpl(lhs), GetImpl(rhs));
         }
   };
}
```

Итак, класс `SmartPtr` не определяет операторы упорядочения по умолчанию. В нем объявляются (но не реализуются) два обобщенных оператора <, а остальные операторы выражаются через них. Пользователь может использовать либо специализированную, либо обобщенную версии оператора <.

Класс `SmartPtr` осуществляет специализацию оператора `std::less`, устанавливая отношение порядка между произвольными интеллектуальными указателями.

7.10. Обнаружение и регистрация ошибок

В разных приложениях требуется разная степень безопасности, обеспечиваемая интеллектуальными указателями. Некоторые программы выполняют интенсивные вычисления, поэтому нужно оптимизировать их быстродействие, другие (фактически большинство) интенсивно осуществляют операции ввода-вывода информации и нуждаются в эффективных средствах проверки данных, не снижающих их производительность.

Чаще всего в приложении нужны обе модели: низкая безопасность, высокая скорость в некоторых критических местах и высокая безопасность, низкая скорость — в остальных частях программы.

Вопросы, связанные с проверкой интеллектуальных указателей, можно разделить на две категории: инициализация проверки и проверка перед разыменованием.

7.10.1. Проверка во время инициализации

Может ли интеллектуальный указатель принимать нулевое значение?

Легко можно сделать так, что интеллектуальный указатель никогда не сможет стать нулевым (очень полезное свойство в практических приложениях). Это значит, что любой интеллектуальный указатель всегда корректен (если вы не манипулируете простыми указателями с помощью функции GetImplRef). Реализовать это свойство интеллектуального указателя можно с помощью конструктора, генерирующего исключительную ситуацию при получении нулевого указателя.

```
template <class T>
class SmartPtr
{
public:
   SmartPtr(T* p) : pointee_(p)
   {
      if (!p) throw NullPointerException();
   }
   ...
};
```

С другой стороны, нулевое значение представляет собой удобное обозначение "неверного указателя" и также часто оказывается полезным.

Разрешение или запрет нулевых значений влияют и на конструктор по умолчанию. Если интеллектуальный указатель не допускает нулевых значений, как конструктор по умолчанию может инициализировать обычный указатель? Применения конструктора по умолчанию можно избежать, но это значительно затрудняет создание интеллектуальных указателей. Например, что вы будете делать, если у вас есть переменная-член класса SmartPtr, но нет подходящей функции, инициализирующей ее во время создания объекта? В заключение отметим, что настройка инициализации содержит выбор подходящих значений, задаваемых по умолчанию.

7.10.2. Проверка перед разыменованием

Проверка перед разыменованием имеет большое значение, поскольку разыменование нулевого указателя приводит к непредсказуемым последствиям. Во многих приложениях это совершенно недопустимо, поэтому проверка корректности указателя перед его разыменованием обязательна. В классе SmartPtr она выполняется операторами -> и *.

В отличие от проверки во время инициализации, проверка перед разыменованием может снизить производительность вашего приложения, поскольку обычно разыменование интеллектуальных указателей выполняется намного чаще, чем их создание. Следовательно, следует стремиться поддерживать баланс между безопасностью и скоростью. Есть хорошее правило: начинать со строгой проверки всех указателей, а затем снимать проверку с некоторых интеллектуальных указателей по совету профилировщика.

Существуют ли принципиальные различия между проверкой во время инициализации и проверкой перед разыменованием? Нет, так как они связаны между собой. Если во время инициализации выполняется строгая проверка, то проверка перед разыменованием становится излишней, поскольку указатели, прошедшие проверку во время инициализации, всегда корректны.

7.10.3. Сообщения об ошибках

Единственный разумный способ регистрировать ошибки — генерировать исключительные ситуации.

Можно предпринять некоторые предосторожности, позволяющие избежать появления ошибок. Например, если указатель перед разыменованием оказался нулевым,

его можно проинициализировать на лету. Эта правильная и ценная стратегия называется *ленивой инициализацией* (lazy initialization) — объект создается только в тот момент, когда он впервые используется.

Если проверка должна выполняться только в процессе отладки программы, можно использовать стандартный макрос `assert` и подобные ему более сложные макросы. В окончательном варианте программы компилятор игнорирует проверки. Следовательно, если все ссылки на нулевые значения во время отладки уже удалены, во время выполнения программы проверку можно не повторять.

В классе `SmartPtr` все проверки сосредоточены в стратегии `Checking`, реализующей соответствующие функции (которые при необходимости могут использовать ленивую инициализацию).

7.11. Интеллектуальные указатели на константные объекты и константные интеллектуальные указатели

Обычные указатели реализуют два вида работы с константами: указатели на константные объекты и собственно константные указатели. Эти свойства указателей иллюстрируются следующим фрагментом программы.

```
const Something* pc = new Something; // Указывает на
                                     // константный объект
pc->ConstMemberFunction();    // Правильно
pc->NonConstMemberFunction(); // Неправильно
delete pc; // Как ни  странно, правильно⁴
Something* const pc = new Something; // Константный указатель
cp->NonConstMemberFunction(); // Теперь правильно
cp = new Something; // Неправильно, константному указателю
                    // ничего нельзя присваивать
const Something* const cpc =
  new Something; // Константный указатель на константный объект
cpc->ConstMemberFunction(); //Правильно
cpc->NonConstMemberFunction(); // Неправильно
cpc = new Something; // Неправильно, константному указателю
                     // ничего нельзя присваивать
```

Соответственно, класс `SmartPtr` используется следующим образом.

```
// Интеллектуальный указатель на константный объект
SmartPtr<const Something> spc(new Something);
// Константный интеллектуальный указатель
const SmartPtr<Something> scp(new Something);
// Константный интеллектуальный указатель на константный объект
const SmartPtr<const Something> scpc(new Something);
```

Шаблонный класс `SmartPtr` может обнаруживать, является ли объект, на который он ссылается, константным, с помощью частичной специализации или шаблонного класса `TypeTraits`, определенного в главе 2. Второй способ более предпочтителен, поскольку он не приводит к дублированию исходного кода, как при частичной специализации.

---

[4] Вопрос "Почему оператор `delete` можно применять к указателям на константные объекты?" всегда вызывает яростные споры в группе новостей comp.std.c++. Однако, хорошо это или плохо, язык допускает такую конструкцию.

Класс SmartPtr имитирует семантику указателей на константные объекты, константных указателей, а также их комбинации.

7.12. Массивы

В большинстве случаев вместо динамических массивов и операторов new[] и delete[] лучше использовать стандартный класс std::vector. Этот класс полностью обеспечивает возможности, предоставляемые динамическими массивами, а также многое другое, почти не тратя при этом дополнительных ресурсов.

Однако "в большинстве случаев" не означает "всегда". Существует много ситуаций, в которых полноценный вектор не нужен и даже не желателен. Именно здесь и нужны динамические массивы. В таких случаях без интеллектуальных указателей обойтись трудно. Между сложным шаблонным класом std::vector и динамическими массивами пролегает довольно глубокая пропасть. Интеллектуальные указатели могут выступать в роли мостиков через эту пропасть, предоставляя пользователю семантические возможности массивов.

С точки зрения интеллектуального указателя на массив единственным важным моментом является использование вызова оператора delete[] pointee_ в его деструкторе вместо оператора delete pointee_. Эта проблема уже решена в стратегии Ownership.

Второй момент связан с индексированным доступом к элементам массива с помощью оператора [], перегруженного для интеллектуальных указателей. Технически это вполне возможно. Фактически в предварительной версии класса SmartPtr для семантики массивов была предусмотрена отдельная стратегия. Однако интеллектуальные указатели очень редко ссылаются на массивы. В этих случаях индексированный доступ можно обеспечить с помощью функции GetImpl.

```
SmartPtr<Widget> sp = ...;
// Доступ к шестому элементу массива, на который ссылается
// интеллектуальный указатель sp
Widget& obj = GetImpl(sp)[5];
```

Было бы нерационально пытаться разработать дополнительные синтаксические конструкции за счет новой стратегии.

Класс SmartPtr поддерживает настраиваемое разрушение объектов с помощью стратегии Ownership. Следовательно, для удаления массивов можно использовать оператор delete[]. Однако класс SmartPtr не поддерживает арифметику указателей.

7.13. Интеллектуальные указатели и многопоточность

Чаще всего интеллектуальные указатели помогают совместно использовать объекты. В свою очередь, многопоточность тесно связана с совместным использованием объектов. Следовательно, многопоточность и интеллектуальные указатели связаны друг с другом.

Взаимодействие между ними проявляется на двух уровнях: на уровне объектов, на которые ссылаются интеллектуальные указатели (pointee object level), и на уровне регистрации данных (bookkeeping data level).

7.13.1. Многопоточность на уровне объектов

Если на один и тот же объект одновременно ссылаются интеллектуальный указатель и несколько потоков, желательно иметь возможность захватывать этот объект на время вызова функции-члена, выполненного с помощью оператора ->. Этого можно

достичь, если интеллектуальный указатель будет возвращать объект-заместитель (proxy object), а не простой указатель. Конструктор объекта-заместителя захватывает объект, на который ссылается интеллектуальный указатель, а его деструктор освобождает его. Этот способ программирования описан в книге Страуструпа (Stroustrup, 2000). Ниже приводится код, иллюстрирующий этот подход.

Во-первых, рассмотрим класс Widget, имеющий две конструкции для захвата объекта: функции-члены Lock и Unlock. После вызова функции Lock открывается безопасный доступ к объекту, при этом вызов такой функции остальными потоками блокируется. После вызова функции Unlock объект может поступать в распоряжение остальных потоков.

```
class Widget
{
   ...
   void Lock();
   void Unlock();
};
```

Затем определим шаблонный класс LockingProxy. Он предназначен для захвата объекта (с помощью описанного выше механизма Lock/Unlock) на время существования объекта класса LockingProxy.

```
template <class T>
class LockingProxy
{
public:
   LockingProxy(T* pObj) : pointee_ (pObj)
   { pointee_->Lock(); }
   ~LockingProxy();
   { pointee_->Unlock(); }
   T* operator->() const
   { return pointee_; }
private:
   LockingProxy& operator=(const LockingProxy&);
   T* pointee;
};
```

Кроме конструктора и деструктора в классе LockingProxy определен оператор ->, возвращающий указатель на объект.

Несмотря на то что класс LockingProxy напоминает интеллектуальный указатель, вокруг него существует еще одна оболочка — сам шаблонный класс SmartPtr.

```
template <class T>
class SmartPtr
{
   ...
   LockingProxy<T> operator->() const;
   { return LockingProxy<T>(pointee_); }
private:
   T* pointee_;
};
```

Напомним, что в разделе 7.3, посвященном механике оператора ->, уже указывалось, что компилятор может применять оператор -> к одному и тому же выражению несколько раз, пока не обнаружит простой указатель. Теперь представьте себе следующий вызов (считая, что в классе Widget определена функция DoSomething).

```
SmartPtr<Widget> sp = ...;
sp->DoSomething();
```

Здесь кроется один нюанс: оператор -> класса SmartPtr возвращает временный объект класа LockingProxy<T>. Компилятор продолжает применять оператор ->. Оператор -> объекта класса LockingProxy<T> возвращает объект, имеющий тип Widget*. Компилятор использует этот указатель на объект класса Widget для вызова функции DoSomething. Во время вызова этой функции объект класса LockingProxy<T> продолжает существовать и владеть объектом, находящимся в полной безопасности. После возврата управления из функции DoSomething временный объект класса LockingProxy<T> уничтожается, и объект класса Widget освобождается.

Автоматический захват объектов — хорошая иллюстрация расслоения интеллектуальных указателей, которое можно осуществить, внеся соответствующие изменения в стратегию Storage.

7.13.2. Многопоточность на уровне регистрации данных

Иногда, кроме объектов, на которые они ссылаются, интеллектуальные указатели манипулируют дополнительными данными. Как указывалось в разделе 7.5, несколько интеллектуальных указателей с подсчетом ссылок скрытно используют один и тот же счетчик. Если скопировать такой указатель из одного потока в другой, возникнут два интеллектуальных указателя, использующих один и тот же счетчик ссылок. Разумеется, они оба продолжают указывать на тот же объект, но пользователю известно, какой из этих указателей в данный момент владеет им. В то же время счетчик ссылок пользователю не доступен и управляется исключительно интеллектуальным указателем.

Для многопоточной среды опасность представляют не только интеллектуальные указатели с подсчетом ссылок. Интеллектуальные указатели с отслеживанием ссылок (раздел 7.5.4) содержат указатели друг на друга, которые также относятся к совместно используемым данным. Связывание ссылок объединяет интеллектуальные указатели в группу, причем они не обязаны принадлежать одному и тому же потоку. Следовательно, каждый раз, когда выполняется копирование, присваивание или уничтожение интеллектуального указателя с отслеживанием ссылок, нужно решать вопрос о захвате объекта, в противном случае дважды связанный список может оказаться разрушенным.

В заключение отметим, что вопросы, связанные с многопоточностью, в итоге влияют на реализацию интеллектуальных указателей. В качестве примера рассмотрим, как многопоточность влияет на подчет ссылок и их связывание.

7.13.2.1. Многопоточный подсчет ссылок

При копировании интеллектуальных указателей из разных потоков счетчик ссылок в разных потоках увеличивается непредсказуемым образом.

Как показано в приложении, операция инкрементации не является атомарной. Для инкрементации и декрементации целых чисел в многопоточной среде нужно использовать тип ThreadingModel<T>::IntType, а также функции AtomicIncrement и AtomicDecrement.

Это все немного усложняет. Вернее, ситуация осложняется, если мы хотим сделать подсчет ссылок независимым от потоков.

Следуя принципам разработки классов на основе стратегий, разложим класс на элементы, описывающие его поведение, и свяжем с каждым из них соответствующий шаблонный параметр. В идеале класс SmartPtr задавал бы стратегии Ownership и ThreadingModel, применяя их для корректной реализации.

Однако при подсчете ссылок в многопоточной среде все намного запутаннее. Например, счетчик должен иметь тип ThreadingModel<T>::IntType. Следовательно,

вместо использования операторов ++ и -- приходится применять функции AtomicIncrement и AtomicDecrement. Потоки и счетчик ссылок сливаются в одно целое, их невероятно трудно разъединить.

Лучше всего включить многопоточность в стратегию Ownership. В этом случае можно получить две реализации: RefCounting и RefCountingMT.

7.13.2.2. Многопоточное связывание ссылок

Рассмотрим деструктор интеллектуального указателя со связыванием ссылок.

```
template <class T>
class SmartPtr
{
public:
   ~SmartPtr()
   {
      if (prev_ == next_)
      {
         delete pointee_;
      }
      else
      {
         prev_->next_ = next_;
         next_->prev_ = prev_;
      }
   }
   ...
private:
   T* pointee_;
   SmartPtr* prev_;
   SmartPtr* next_;
};
```

Деструктор выполняет классическое удаление элемента из дважды связанного списка. Для упрощения и ускорения реализации список сделан кольцевым — последний узел ссылается на первый. Это позволяет не проверять, являются ли указатели prev_ и next_ нулевыми. В кольцевом списке, состоящем из одного элемента, указатели prev_ и next_ будут равны указателю this.

Если несколько потоков уничтожают интеллектуальные указатели, связанные друг с другом, деструктор, очевидно, должен быть атомарным (т.е. его работу не могут прервать другие потоки). В противном случае работу деструктора класса SmartPtr сможет прервать любой поток. Например, это может произойти в момент между обновлением указателей prev_ и next_. В этом случае поток, прервавший работу деструктора, будет оперировать разрушенным списком.

Аналогичные рассуждения касаются и конструктора копирования, и оператора присваивания класса SmartPtr. Эти функции должны быть атомарными, поскольку они манипулируют со списком владения.

Интересно, что здесь нельзя применить семантику захвата объектов. В приложении, приведенном в конце книги, стратегии захвата разделены на *стратегии уровня классов* (class level strategies) и *стратегии уровня объектов* (object-level strategies). Операции захвата на уровне классов захватывают все объекты данного класса. В то же время операции захвата на уровне объектов относятся только к одному объекту. В первом случае тратится меньше памяти (только один мьютекс на класс), но возникают проблемы с производительностью программы. Второй случай (один мьютекс на объект) сложнее, но позволяет создавать реализации, которые работают быстрее.

К интеллектуальным указателям нельзя применять стратегию захвата на уровне объектов, поскольку операция копирования манипулирует сразу тремя объектами: текущий объект, подлежащий вставке или удалению, предыдущий объект и следующий объект в списке владения.

Если все же возникает необходимость реализовать захват на уровне объектов, следует убедиться, что каждому объекту соответствует один мьютекс, поскольку для каждого объекта существует отдельный список владения. Мьютексы для каждого объекта можно разместить в динамической памяти, хотя это сводит к нулю все преимущества связывания ссылок над их подсчетом. Связывание ссылок привлекательно именно потому, что оно не использует динамическую память.

В качестве альтернативы можно использовать стратегию внедрения: мьютекс хранится в объекте, на который ссылается интеллектуальный указатель. Однако существование разумной и эффективной альтернативы — интеллектуальных указателей с подсчетом ссылок — вынуждает отказаться от реализации этой стратегии.

Итак, интеллектуальные указатели, использующие подсчет ссылок или их связывание, возникают в многопоточной среде. Для реализации безопасной стратегии подсчета ссылок необходимы атомарные операции. Для реализации безопасной стратегии связывания ссылок нужны мьютексы. Класс `SmartPtr` реализует только безопасную стратегию подсчета ссылок.

7.14. Сборка

До сих пор мы рассматривали каждый вопрос по отдельности. Настало время собрать все решения воедино и воплотить их в реализации класса `SmartPtr`.

Мы будем по-прежнему следовать принципам разработки классов на основе стратегий, описанным в главе 1. Каждый аспект проектирования, не имеющий единственного решения, реализуется в классах стратегий. Шаблонный класс `SmartPtr` получает каждую стратегию в виде отдельного шаблонного параметра, наследует все эти шаблонные параметры, позволяя соответствующим стратегиям сохранять состояние.

Вернемся мысленно к предыдущим разделам, перечисляя основные аспекты класса `SmartPtr`, каждый из которых представляет собой отдельную стратегию.

- *Стратегия* `Storage` *(раздел 7.3).* По умолчанию сохраняемым типом является тип `T*` (тип `T` — это первый шаблонный параметр класса `SmartPtr`), типом указателей — тоже `T*`, а ссылочным типом — `T&`. Объект, на который ссылается интеллектуальный указатель, уничтожается оператором `delete`.
- *Стратегия* `Ownership` *(раздел 7.5).* Обычно реализуется с помощью глубокого копирования, подсчета ссылок, связывания ссылок и разрушающего копирования. Обратите внимание на то, что стратегия `Ownership` не связана с механизмом уничтожения объектов, который относится к стратегии `Storage`. Стратегия `Ownership` лишь указывает *момент* уничтожения объекта.
- *Стратегия* `Conversion` *(раздел 7.7).* В некоторых приложениях необходимо преобразовывать интеллектуальные указатели в обычные. Обратное преобразование не допускается.
- *Стратегия* `Checking` *(раздел 7.10).* Эта стратегия проверяет правильность инициализации и разыменования класса `SmartPtr`.

Остальные свойства не настолько важны, чтобы создавать для них отдельные стратегии.

- Оператор взятия адреса (раздел 7.6) лучше не перегружать.

- Проверка равенства и неравенства выполняется с помощью приема, описанного в разделе 7.8.
- Отношения порядка (раздел 7.9) остаются нереализованными. Однако в библиотеке Loki осуществляется специализация функции `std::less` для объектов класса `SmartPtr`. Пользователь может определить оператор <, а библиотека Loki поможет выразить через него остальные операторы сравнения.
- В библиотеке Loki определена корректная реализация константных объектов класса `SmartPtr`, объектов, на которые ссылаются интеллектуальные указатели, а также их обоих одновременно.
- Массивы специально не предусматриваются, однако одна из готовых реализаций стратегии `Storage` может удалять массивы с помощью оператора `delete[]`.

Каждый аспект реализации класса `SmartPtr` рассматривался отдельно от остальных. Это позволяет лучше разобраться в их механизмах. Намного полезнее разобрать реализацию на части и рассмотреть их по отдельности, чем изучать их в комплексе.

Разделяй и властвуй — этот старый девиз Юлия Цезаря вполне пригоден для разработки интеллектуальных указателей. (Бьюсь об заклад, он этого не предвидел!) Мы разбиваем проблемы на небольшие составные классы, называемые *стратегиями* (policy). Каждая стратегия связана только с одним аспектом реализации класса. Класс `SmartPtr` наследует все эти классы, одновременно приобретая все их свойства. Это очень простой и невероятно гибкий механизм. Кроме того, каждая стратегия задается одним шаблонным параметром. Это позволят смешивать и сопоставлять существующие классы стратегий, а также создавать на их основе свои собственные стратегии.

Первым идет тип объекта, на который ссылается интеллектуальный указатель, за ним — все стратегии. В итоге получается следующее объявление класса `SmartPtr`.

```
template
<
   typename T,
   template <class> class OwnershipPolicy = RefCounted,
   class ConversionPolicy = DisallowConversion,
   template <class> class CheckingPolicy = AssertCheck,
   template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

Сначала следует указывать стратегии, которые настраиваются чаще других.

Ниже рассматриваются требования, предъявляемые к четырем стратегиям, определенным ранее. Все эти стратегии должны иметь семантику значений, т.е. они должны определять соответствующие конструктор копирования и оператор присваивания.

7.14.1. Многопоточность на уровне объектов

Стратегия `Storage` абстрагирует структуру интеллектуального указателя. Она осуществляет определение типов и хранит фактический объект `pointee_`.

Если класс `StorageImpl` является реализацией стратегии `Storage`, а `storageImpl` — это объект типа `StorageImpl<T>`, то применяются конструкции, указанные в табл. 7.1.

Реализация стратегии `Storage` по умолчанию имеет следующий вид.

```
template <class T>
class DefaultSPStorage
{
protected:
```

```
    typedef T* StoredType;    // Тип объекта pointee_
    typedef T* PointerType;   // Тип объекта,
                              // возвращаемого оператором ->
    typedef T& ReferenceType; // Тип объекта,
                              // возвращаемого оператором *
public:
    DefaultSPStorage() : pointee_(Default())
{}
    DefaultSPStorage(const StoredType& p): pointee_(p) {}
    PointerType operator->() const { return pointee_; }
    ReferenceType operator*() const { return *pointee_; }
    friend inline PointerType GetImpl(const DefaultSPStorage& sp)
    { return sp.pointee_; }
    friend inline const StoredType& GetImplRef(
      const DefaultSpStorage& sp)
    { return sp.pointee_; }
    friend inline StoredType& GetImplRef(DefaultSPStorage& sp)
    { return sp.pointee_; }
protected:
    void Destroy()
    { delete pointee_; }
    static StoredType Default()
    { return 0; }
private:
    StoredType pointee_;
};
```

Кроме класса DefaultSPStorage в библиотеке Loki определены следующие классы.

- Класс ArrayStorage, использующий оператор delete[] внутри функции Release.
- Класс LockedStorage, использующий иерархическую реализацию интеллектуального класса, захватывающего данные при разыменовании (раздел 7.13.1).
- Класс HeapStorage, использующий явный вызов деструктора и функцию std::free для освобождения данных.

Таблица 7.1. Конструкции стратегии Storage

Выражение	Семантика
StorageImpl<T>::StoredType	Тип, фактически хранимый реализацией. По умолчанию: T*
StorageImpl<T>::PointerType	Тип указателей, определенный реализацией. Возвращается оператором ->, определенным в классе SmartPtr. По умолчанию: T*. Может отличаться от типа StorageImpl<T>::StoredType, если используется иерархия интеллектуальных указателей (разделы 7.3 и 7.13.1)
StorageImpl<T>::ReferenceType	Ссылочный тип. Возвращается оператором * класса SmartPtr. По умолчанию: T&
GetImpl(storageImpl)	Возвращает объект типа StorageImpl<T>::StoredType
GetImplRef(storageImpl)	Возвращает объект типа StorageImpl<T>::StoredType& (если объект storageImpl является константным, конструкция объявляется константной)

Выражение	Семантика
`storageImpl.operator->()`	Возвращает объект типа `StorageImpl<T>::PointerType`. Используется оператором `->` класса `SmartPtr`
`storageImpl.operator*()`	Возвращает объект типа `StorageImpl<T>::ReferenceType`. Используется оператором `*` класса `SmartPtr`
`StorageImpl<T>::StoredType p;` `p = storageImpl.Default();`	Возвращает значение, заданное по умолчанию (обычно нуль)
`storageImpl.Destroy()`	Разрушает объект, на который ссылается интеллектуальный указатель

7.14.2. Стратегия Ownership

Стратегия `Ownership` должна поддерживать как внедренный, так и независимый подсчет ссылок. Следовательно, она должна использовать явные вызовы функции, а не механизм конструкторов и деструкторов, как это сделано в работе (Koenig, 1996). Дело в том, что функцию-член можно вызвать в любое время, а конструкторы и деструкторы вызываются автоматически и только в определенные моменты времени.

Реализация стратегии `Ownership` имеет один шаблонный параметр, соответствующий ссылочному типу. Класс `SmartPtr` передает параметр `StoragePolicy<T>::PointerType` классу `OwnershipPolicy`. Обратите внимание на то, что шаблонный параметр класса `OwnershipPolicy` является типом указателя, а не объекта, на который он ссылается.

Если класс `OwnershipImpl` является реализацией стратегии `Ownership`, а `ownershipImpl` — это объект типа `OwnershipImpl<P>`, то применяются конструкции, указанные в табл. 7.2.

Таблица 7.2. Конструкции стратегии Ownership

Выражение	Семантика
`P val1;` `P val2 = OwnershipImpl.Clone(val1);`	Клонирует объект. Может модифицировать исходное значение, если в стратегии `Ownership` применяется разрушающее копирование
`const P val1;` `P val2 = ownershipImpl.Clone(val1);`	Клонирует объект
`P val;` `bool unique = ownershipImpl.Release(val);`	Освобождает объект. Возвращает значение `true`, если была освобождена последняя ссылка на объект
`bool dc = OwnershipImpl<P>::destructiveCopy;`	Устанавливает, должна ли стратегия `Ownership` применять разрушающее копирование. Если да, класс `SmartPtr` применяет примем Колвина–Гиббонса (Meyers, 1999), использованный в классе `std::auto_ptr`

Реализация стратегии Ownership, основанной на подсчете ссылок, выглядит следующим образом.

```
template <class P>
class RefCounted
{
   unsigned int* pCount_;
protected:
   RefCounted() : pCount_(new unsigned int(1)) {}
   P Clone(const P & val)
   {
      ++*pCount_;
      return val;
   }
   bool Release(const P&)
   {
      if (!--*pCount_)
      {
          delete pCount_;
          return true;
      }
      return false;
   }
   enum { destructiveCopy = false } ; // см. ниже
};
```

Реализовать эту стратегию на основе других схем подсчета ссылок так же просто. Рассмотрим реализацию стратегии Ownership для COM-объектов. Эти объекты имеют две функции: Addref и Release. При последнем вызове функции Release объект разрушается. Нужно лишь передать адрес функции Clone в качестве параметра функции AddRef и адрес функции Release в качестве параметра функции Release, принадлежащей COM-объекту.

```
template <class P>
class COMRefCounted
{
public:
   static P Clone(const P& val)
   {
      val->AddRef();
      return val;
   }
   static bool Release(const P& val)
   {
      val->Release();
      return false;
   }
   enum { destructiveCopy = false }; // см. ниже
};
```

В библиотеке Loki определены следующие реализации стратегии Ownership.

- Класс DeepCopy, описанный в разделе 7.5.1. В нем предполагается, что в классе, на объекты которого ссылается указатель (pontee class), реализована функция-член Clone.
- Класс RefCounted, описанный в разделе 7.5.3 и в данном разделе.
- Класс RefCountedMT, многопоточная версия класса RefCounted.
- Класс ComRefCounted — вариант внедренного подсчета ссылок, описанный в данном разделе.

- Класс RefLinked, описанный в разделе 7.5.4.
- Класс DestructiveCopy, описанный в разделе 7.5.5.
- Класс NoCopy, который не определяет функцию Clone, блокируя любой вид копирования.

7.14.3. Стратегия Conversion

Это довольно простая стратегия: она определяет булевскую статическую константу, значение которой зависит от того, разрешено в классе SmartPtr неявное преобразование в базовый тип указателя или нет.

Если класс ConversionImpl является реализацией стратегии Conversion, применяются конструкции, приведенные в табл. 7.3.

Базовый тип указателя класса SmartPtr определяется стратегией Storage и классом StorageImpl<T>::PointerType.

Как и следовало ожидать, в библиотеке Loki точно определены две реализации стратегии Convesrion.

- Класс AllowConversion.
- Класс DisallowConversion.

Таблица 7.3. Конструкции стратегии Conversion

Выражение	Семантика
`bool allowConv = ConversionImpl<P>::allow;`	Если константа allow имеет значение true, класс SmartPtr допускает неявное преобразование в базовый тип указателя

7.14.4. Стратегия Checking

Как указывалось в разделе 7.10, проверять объекты класса SmartPtr можно во время инициализации и перед разыменованием. Во время проверки можно использовать макрос assert, исключительные ситуации, ленивую инициализацию или вообще ничего не делать.

Стратегия Checking оперирует классом StoredType, принадлежащим стратегии Storage, а не типом PointerType. (Определение стратегии Storage дано в разделе 7.14.1.)

Если S — это сохраняемый тип, определенный в реализации стратегии Storage, класс CheckImpl реализует стратегии Checking, а checkingImpl — это объект типа CheckingImpl<S>, то применяются конструкции, указанные в табл. 7.4.

Таблица 7.4. Конструкции стратегии Checking

Выражение	Семантика
`S value; checkingImpl.OnDefault(value);`	Объект класса SmartPtr вызывает функцию OnDefault при вызове конструктора по умолчанию. Если в классе CheckingImpl эта функция не определена, конструктор по умолчанию блокируется на этапе компиляции
`S value; checkingImpl.OnInit(value);`	Объект класса SmartPtr вызывает функцию OnInit при вызове конструктора

Окончание табл. 7.4

Выражение	Семантика
`S value; checkingImpl.OnDereference (value);`	Объект класса `SmartPtr` вызывает функцию `OnDereference` перед возвращением результата выполнения операторов `->` и `*`
`const S value; checkingImpl.OnDereference (value);`	Объект класса `SmartPtr` вызывает функцию `OnDereference` перед возвращением результата выполнения константных версий операторов `->` и `*`

В библиотеке Loki определены следующие реализации стратегии `Checking`.

- Класс `AssertCheck`, использующий макрос `assert` для проверки значения перед разыменованием.
- Класс `AssertCheckStrickt`, использующий макрос `assert` для проверки значения при инициализации.
- Класс `RejectNullStatic`, не определяющий функцию `OnDefault`. Следовательно, любая попытка вызвать конструктор по умолчанию в классе `SmartPtr` приведет к появлению ошибки компиляции.
- Класс `RejectNull`, возбуждающий исключительную ситуацию при попытке разыменования нулевого указателя.
- Класс `RejectNullStrict`, не позволяющий использовать нулевой указатель в качестве начального значения (генерируя при этом исключительную ситуацию).
- Класс `NoCheck`, обрабатывающий ошибки в лучших традициях языков С и С++, т.е. не проверяющий их совсем.

7.15. Резюме

Поздравляем! Вы добрались до конца самой длинной и обширной главы. Надеемся, ваши усилия были не напрасны. Теперь вы многое знаете об интеллектуальных указателях и вооружены лаконичным и легко настраиваемым шаблонным классом `SmartPtr`.

Интеллектуальные указатели имитируют синтаксис и семантику простых указателей. Кроме того, они решают многие задачи, выходящие за пределы возможностей простых указателей. Эти задачи могут включать управление владением и проверку значений.

Концепция интеллектуальных указателей намного шире простых указателей. Она может быть обобщена до уровня интеллектуальных ресурсов, например, моникеров (дескрипторов, напоминающих указатели, но имеющих иной синтаксис).

Поскольку интеллектуальные указатели легко автоматизируют процессы, которые практически невозможно реализовать вручную, они стали важным компонентом современных приложений, от которых требуется особая надежность. Именно от них зависит, будет программа выполнена успешно или ее работа приведет к утечке ресурсов.

Вот почему разработчики интеллектуальных указателей должны уделять им особое внимание. Со своей стороны, программисты, использующие интеллектуальные указатели, должны хорошо понимать правила работы с ними и точно им следовать.

Представляя реализацию интеллектуальных указателей, мы уделяли основное внимание вопросу декомпозиции функциональных возможностей на независимые стратегии, которые можно сопоставлять и смешивать в классе `SmartPtr`. Возможно поэтому, каждая стратегия реализует тщательно продуманный интерфейс.

7.16. Краткий обзор класса SmartPtr

- Объявление класса SmartPtr

```
template
<
 typename T,
 template <class> class OwnershipPolicy = RefCounted,
 class ConversionPolicy = DisallowConversion,
 template <class> class CheckingPolicy = AssertCheck,
 template <class> class StoragePolicy = DefaultSPStorage
>
class SmartPtr;
```

- T — это тип, на который указывает объект класса SmartPtr. Тип T может быть основным или определяться пользователем. Допускается использование типа void.
- Для оставшихся параметров шаблонного класса (OwnershipPolicy, ConversionPolicy, CheckingPolicy и StoragePolicy) можно реализовать свои собственные стратегии или выбирать их по умолчанию, как показано в разделах 7.14.1–7.14.4.
- Шаблонный параметр OwnershipPolicy задает стратегию владения. В качестве такой стратегии можно выбирать классы DeepCopy, RefCounted, RefCountedMT, COMRefCounted, RefLinked, DestructiveCopy и NoCopy, описанные в разделе 7.14.2.
- Шаблонный параметр ConversionPolicy устанавливает, допускается ли неявное преобразование в базовый тип указателя. По умолчанию неявное преобразование запрещено. В любом случае доступ к объекту осуществляется с помощью вызова функции GetImpl. В качестве реализации стратегии можно использовать классы AllowConversion и DisallowConversion (раздел 7.14.3).
- Шаблонный параметр CheckingPolicy задает стратегию проверки ошибок. По умолчанию можно использовать классы AssertCheck, AssertCheckStrict, RejectNullStatic, RejectNullStrict и NoCheck (раздел 7.14.4).
- Шаблонный параметр StoragePolicy определяет механизм хранения и доступа к объекту. По умолчанию применяется класс DefaultSPStorage, который, будучи конкретизирован типом T, определяет ссылочный тип T&, сохраняемый тип T*, а также тип T*, возвращаемый оператором ->. В библиотеке Loki определены также сохраняемые типы ArrayStorage, LockedStorage и HeapStorage (раздел 7.14.1).

8

Фабрики объектов

Для достижения высокого уровня абстракции и модульности в объектно-ориентированных программах применяются механизм наследования и виртуальные функции. Полиморфизм, предоставляющий возможность отложить выбор вызываемой функции на период выполнения программы, обеспечивает повторное использование бинарного кода и его адаптацию. Система поддержки выполнения программ автоматически распределяет виртуальные функции по соответствующим объектам производных классов, позволяя описать сложное поведение с помощью полиморфных примитивов.

Параграф, приведенный выше, можно найти в любой книге, посвященной объектно-ориентированному программированию. Мы цитируем эти высказывания для того, чтобы проиллюстрировать контраст между благополучным состоянием дел в "стационарном режиме" и сложной ситуацией, складывающейся в "режиме инициализации", когда объект нужно *создавать* с помощью полиморфных методов.

В стационарном режиме у нас уже есть указатели или ссылки на полиморфные объекты, и мы можем вызывать соответствующие функции-члены. Их динамический тип хорошо известен (хотя вызывающий модуль может его не знать). Однако существуют случаи, когда при создании объектов нужна такая же гибкость. Это приводит к парадоксу "виртуальных конструкторов". Виртуальные конструкторы необходимы, когда информация о создаваемом объекте носит динамический характер и не может быть использована в конструкциях языка C++ непосредственно.

Чаще всего полиморфные объекты создаются в динамической памяти с помощью оператора new.

```
class Base { ... };
class Derived : public Base { ... };
class AnotherDerived : public Base { ... };
...
// Создаем объект класса Derived и присваиваем его адрес
// указателю на класс Base
Base* pB = new Derived;
```

Проблема возникает оттого, что фактический тип Derived указывается непосредственно в вызове оператора new. Между прочим, класс Derived здесь играет ту же роль, что и явно заданные числовые константы, которых следует избегать. Если нужно создать объект типа AnotherDerived, приходится изменять соответствующий оператор и изменять тип Derived на AnotherDerived. Сделать этот процесс динамическим невозможно: оператору new можно передавать только типы, точно известные во время компиляции.

В этом заключается принципиальная разница между созданием объектов и вызовом виртуальных функций-членов в языке C++. Виртуальные функции-члены явля-

ются динамическими — их поведение можно изменять, не модифицируя вызывающий код. В противоположность этому, каждое создание объекта — это камень преткновения для статического, жестко определенного кода. В результате вызовы виртуальных функций связывают вызывающий код только с интерфейсом (базовым классом). Объектно-ориентированное программирование стремится снять ограничения, накладываемые необходимостью указывать фактический тип создаваемого объекта. Однако, по крайней мере в языке C++, создание объектов по-прежнему связывает вызывающий код с конкретным классом, находящимся на самом дне иерархиии.

На самом деле эта проблема намного глубже: даже в повседневной жизни создать нечто и пользоваться им — совершенно разные вещи. Обычно предполагается, что, создавая определенный объект, вы точно знаете, что будете с ним делать. И все же иногда в этом правиле возникают исключения. Это происходит в следующих ситуациях.

- Если точное знание о каком-то объекте нужно передать другой сущности. Например, вместо непосредственного вызова оператора `new` можно вызвать виртуальную функцию `Create`, принадлежащую объекту более высокого иерархического уровня, позволяя пользователям изменять его поведение с помощью полиморфизма.
- Если знания об объекте не могут быть выражены с помощью средств языка C++. Например, пусть задана строка `"Derived"`. В этом случае программист точно знает, что нужно создать объект типа `Derived`, однако эту строку невозможно передать оператору `new` в качестве имени типа.

Для решения этих принципиально важных задач предназначены фабрики объектов. В этой главе обсуждаются следующие темы.

- Ситуации, в которых необходимы фабрики объектов.
- Почему виртуальные конструкторы трудно реализовать в языке C++.
- Как создавать объекты, задавая их типы в виде значений.
- Реализация обобщенной фабрики объектов.

В конце главы мы построим обобщенную фабрику объектов. Ее можно настраивать по типу, способу создания и методу идентификации объектов. Фабрику объектов можно использовать в сочетании с другими компонентами, описанными в этой книге, например, синглтонами (глава 6) — для создания фабрики объектов внутри приложения и функторами (глава 5) — для настройки работы фабрики. Мы рассмотрим фабрику клонов, которая может создавать дубликаты объектов любого типа.

8.1. Для чего нужны фабрики объектов

Фабрики объектов нужны в двух случаях. Во-первых, когда библиотека должна не только манипулировать готовыми объектами, но и создавать их. Например, представьте себе, что мы разрабатываем интегрированную среду для многооконного текстового редактора. Поскольку эта среда должна быть легко расширяемой, мы предусматриваем абстрактный класс `Document`, на основе которого пользователи могут создавать производные классы `TextDocument` и `HTMLDocument`. Другой компонент интегрированной среды — класс `DocumentManager`, хранящий список всех открытых документов.

Следует установить правило: каждый документ, существующий в приложении, должен быть известен классу `DocumentManager`. Следовательно, создание нового документа тесно связано с его вставкой в список документов, хранящийся в классе

DocumentManager. Когда две операции настолько тесно связаны, лучше всего описать их с помощью одной и той же функции и никогда не выполнять по отдельности.

```
class DocumentManager
{
   ...
public:
   Document* NewDocument();
private:
   virtual Document* CreateDocument() = 0;
   std::list<Document*> listOfDocs_;
};

Document* DocumentManager::NewDocument()
{
   Document* pDoc = CreateDocument();
   listOfDocs_.push_back(pDoc);
   ...
   return pDoc;
}
```

Функция-член CreateDocument заменяет вызов оператора new. Функция-член NewDocument не может использовать оператор new, поскольку при написании класса DocumentManager еще неизвестно, какой конкретно документ будет создан. Для того чтобы использовать интегрированную среду, программист создает производный класс на основе класса DocumentManager и замещает виртуальную функцию-член CreateDocument (которая должна быть чисто виртуальной). В книге GoF (Gamma et al., 1995) метод CreateDocument называется *фабрикой* (factory method).

Поскольку в производном классе точно известен тип создаваемого метода, оператор new можно вызывать непосредственно. Таким образом, в интегрированной среде не обязательно хранить информацию о типе создаваемого документа и можно оперировать только базовым классом Document. Замещение функций-членов осуществляется очень просто и, по сути, сводится к вызову оператора new.

```
Document* GraphicDocumentManager::CreateDocument()
{
   return new GraphicDocument;
}
```

В то же время приложение, построенное на основе этой интегрированной среды, может поддерживать создание нескольких типов документов (например, растровую и векторную графику). В этом случае замещенная функция-член CreateDocument может выводить на экран диалоговое окно, предлагая пользователю ввести конкретный тип создаваемого документа.

Открытие документа, ранее сохраненного на диске, создает новую и более сложную проблему, которую можно решить с помощью фабрики объектов. Записывая объект в файл, его фактический тип следует сохранять в виде строки, целого числа или какого-нибудь идентификатора. Таким образом, хотя информация о типе существует, форма ее хранения не позволяет создавать объекты языка C++.

Эта задача носит общий характер и связана с созданием объектов, тип которых определяется во время выполнения программы: вводится пользователем, считывается из файла, загружается из сети и т.п. В этом случае связь между типом и значениями еще глубже, чем в полиморфизме: используя полиморфизм, сущность, манипулирующая объектом, ничего не знает о его типе; однако сам объект имеет точно определенный тип. При считывании объектов из места их постоянного хранения тип отрывается от

объекта и должен сам преобразовываться в некий объект. Кроме того, считывать объект из места его хранения следует с помощью виртуальной функции.

Создание объектов из "чистой" информации о типе и последующее преобразование *динамической* информации в *статические* типы языка C++ — важная проблема при разработке фабрик объектов. Она рассматривается в следующем разделе.

8.2. Фабрики объектов в языке C++: классы и объекты

Чтобы найти решение проблемы, нужно хорошо в ней разобраться. В этом разделе мы попытаемся найти ответы на следующие вопросы. Почему конструкторы в языке C++ имеют такие жесткие ограничения? Почему в самом языке нет гибких средств для создания объектов?

Поиск ответа на эти вопросы приводит нас к основам системы типов языка C++. Рассмотрим следующий оператор.

```
Base* pB = new Derived;
```

Для того чтобы понять, почему он имеет настолько жесткие ограничения, нам нужно ответить на два вопроса. Что такое класс и что такое объект? Эти вопросы вызваны тем, что основной причиной неудобств в приведенном выше операторе является имя класса `Derived`, которое требуется явно задавать в операторе `new`, хотя мы бы предпочли представить это имя в виде значения, т.е. объекта.

В языке C++ классы и объекты — совершенно разные сущности. Классы создаются программистом, а объекты — программой. Новый класс невозможно создать во время выполнения программы, а объект невозможно создать во время компиляции. Классы не имеют статуса: их нельзя копировать, хранить в переменной или возвращать из функции.

Существуют языки, в которых классы являются *объектами*. В этих языках некоторые объекты с определенными свойствами по умолчанию рассматриваются как классы. Следовательно, в этих языках во время выполнения программы *можно* создавать новые классы, копировать их, хранить в переменных и т.д. Если бы язык C++ был одним из таких языков, можно было бы написать примерно такой код.

```
// Предупреждение — это НЕ C++
// Будем считать, что Class — это класс и объект одновременно
Class Read(const char* fileName);
Document* DocumentManager::OpenDocument(const char* fileName)
{
   Class theClass = Read(fileName);
   Document* pDoc = new theClass;
   ...
}
```

Таким образом, переменную типа `Class` можно было бы передавать оператору `new`. В рамках этой парадигмы передача известного имени класса оператору `new` ничем не отличалась бы от явного указания константы.

В таких динамических языках за счет потери гибкости достигнут определенный компромисс между типовой безопасностью и эффективностью, поскольку статическая типизация — это важный компонент оптимизации кода. В языке C++ принят прямо противоположный подход — опора на статическую систему типов, позволяющая достигать максимальной гибкости.

Итак, создание фабрик объектов в языке C++ представляет собой сложную задачу. В языке C++ между типами и значениями лежит пропасть: значение имеет тип, но

тип не может существовать сам по себе. Для создания объекта чисто динамическим способом нужно иметь средства для выражения "чистого" типа, передачи его во внешнюю среду и создания значения на основе этой информации. Поскольку это невозможно, нужно как-то выразить типы с помощью объектов — целых чисел, строк и т.п. Затем следует придумать способы идентификации типов и создания на их основе соответствующих объектов. Формула объект-тип-объект лежит в основе создания фабрик объектов в языках со статической типизацией.

Объект, идентифицирующий тип, мы будем называть *идентификатором типа* (type identifier) (Не путайте его с ключевым словом typeid.) Идентификатор типа позволяет фабрике объектов создавать соответствующий тип. Как мы увидим впоследствии, иногда идентификатор типа можно создавать, ничего не зная о том, что у нас есть и что мы получим. Это похоже на сказку: вы не знаете точно, что означает заклинание (и пытаться это узнать бывает небезопасно), но произносите его, и волшебник возвращает вам полезную вещь. Детали этого превращения знает только волшебник..., т.е. фабрика.

Мы построим простую фабрику, решающую конкретную задачу, рассмотрим ее разные реализации, а затем выделим ее обобщенную часть в шаблонный класс.

8.3. Реализация фабрики объектов

Допустим, что мы создаем приложение для рисования, позволяющее редактировать простые векторные рисунки, состоящие из линий, окружностей, многоугольников и т.п.[1] В рамках классической объектно-ориентированной парадигмы мы определяем абстрактный класс Shape, из которого будут выведены все остальные фигуры.

```
class Shape
{
public:
   virtual void Draw() const = 0;
   virtual void Rotate(double angle) = 0;
   virtual void Zoom(double zoomFactor) = 0;
   ...
};
```

Затем можно определить класс Drawing, содержащий сложный рисунок. По существу, этот класс хранит набор указателей на объекты класса Shape, т.е. список, вектор или иерархическую структуру, и обеспечивает выполнение операций рисования в целом. Разумеется, нам понадобится сохранять рисунки в файл и загружать их оттуда.

Сохранить фигуру просто: нужно лишь предусмотреть виртуальную функцию Shape::Save(std::ostream&). Операция Drawing::Save может выглядеть следующим образом.

```
class Drawing
{
public:
   void Save(std::ofstream& outFile);
   void Load(std::ifstream& inFile);
   ...
};

void Drawing::Save(std::ofstream& outFile)
```

[1] Эта разработка в духе "Здравствуй, мир!" — хорошая основа для проверки знания языка C++. Многие пробовали в ней разобраться, но лишь некоторые из них поняли, как объект загружается из файла, что, в общем-то, не самое важное.

```
{
    записываем заголовок рисунка
    for (для каждого элемента рисунка)
    {
        (текущий элемент)->Save(outFile);
    }
}
```

Описанный выше пример часто встречается в книгах, посвященных языку С++, включая классическую книгу Б. Страуструпа (Stroustrup, 1997). Однако в большинстве книг не рассматривается операция загрузки рисунка из файла, потому что она разрушает целостность этой прекрасной на вид модели. Углубление в детали загрузки рисунка заставило бы авторов написать большое количество скобок, нарушив гармонию. С другой стороны, именно эту операцию мы хотим реализовать, поэтому нам придется засучить рукава. Проще всего потребовать, чтобы в начале каждого объекта класса, производного от класса Shape, хранился целочисленный уникальный идентификатор. Тогда код для считывания рисунка из файла выглядел бы следующим образом.

```
// Уникальный ID для каждого типа изображаемого объекта
namespace DrawingType
{
const int
    LINE = 1,
    POLYGON = 2,
    CIRCLE = 3
};

void Drawing::Load(std::ifstream& inFile)
{
    // Обработка ошибок для простоты пропускается
    while (inFile)
    {
        // Считываем тип объекта
        int drawingType;
        inFile >> drawingType;

        // Создаем новый пустой объект
        Shape* pCurrentObject;
        switch (drawingType)
        {
            using namespace DrawingType;
        case LINE:
            pCurrentObject = new Line;
            break;
        case POLYGON:
            pCurrentObject = new Polygon;
            break;
        case CIRCLE:
            pCurrentObject = new Circle;
            break;
        default:
            обработка ошибки — неизвестный тип объекта
        }
        // Считываем содержимое объекта,
        // вызывая виртуальную функцию fn
        pCurrentObject->Read(inFile);
        добавляем объект в контейнер
    }
}
```

Это настоящая фабрика объектов. Она считывает из файла идентификатор типа, основываясь на этом идентификаторе, создает объект соответствующего типа и вызывает виртуальную функцию, загружающую этот объект из файла. Одно плохо: все это нарушает важные правила объектно-ориентированного программирования.

- Выполняется оператор switch, основанный на метке типа со всеми вытекающими отсюда последствиями. Именно это категорически запрещено в объектно-ориентированных программах.
- В одном файле концентрируется информация обо всех классах, производных от класса Shape, что также нежелательно. Файл реализации функции Drawing::Save вынужден содержать все заголовки всех возможных фигур. Это делает его уязвимым и зависимым от компилятора.
- Класс трудно расширить. Представьте себе, что нам понадобилось добавить новую фигуру, скажем, класс Ellipse. Кроме создания нового класса, придется добавить новую целочисленную константу в пространство имен DrawingType, записать ее при сохранении объекта класса Ellipse и добавить новую метку в оператор switch внутри функции-члена Drawing::Save. И все это ради одной-единственной функции!

Попробуем создать фабрику объектов, лишенную указанных недостатков. Для этого придется отказаться от использования оператора switch, чтобы мы могли выполнять операции по созданию объектов классов Line, Polygon и Circle единообразно.

Для того чтобы связать между собой фрагменты кода, лучше всего использовать указатели на функции, описанные в главе 5. Фрагмент настраиваемого кода (каждый из элементов оператора switch) можно абстрагировать с помощью следующей сигнатуры.

```
Shape* CreateConcreteShape();
```

Фабрика хранит набор указателей на функции вместе с сигнатурами. Кроме того, нужно установить соответствие между идентификатором типа и указателем на функцию, создающую объект. Таким образом, нам нужен ассоциативный массив (map), предоставляющий доступ к соответствующей функции по идентификатору типа (именно то, что делал оператор switch). Кроме того, этот массив гарантирует масштабируемость, чего оператор switch с его фиксированной статической структурой обеспечить не может. Во время выполнения программы ассоциативный массив может возрастать — мы можем динамически добавлять в него новые элементы (кортежи, состоящие из идентификаторов типов и указателей на функции), а именно это нам и нужно. Можно начать с пустого массива, а затем каждый объект класса, производного от класса Shape, добавит в него новый элемент.

Почему нельзя использовать вектор? Идентификаторы типов являются целыми числами, поэтому их можно считать индексами вектора. Это проще и быстрее, но ассоциативный массив все же лучше. Индексы в таком массиве не обязаны быть смежными. Кроме того, в векторе индексы могут быть только целыми числами, в то время как индексом ассоциативного массива может быть любой тип, для которого установлено отношение порядка. При обобщении нашего примера этот факт приобретает особое значение.

Начнем с разработки класса ShapeFactory, предназначенного для создания всех объектов классов, производных от класса Shape. В реализации класса ShapeFactory мы будем использовать ассоциативный массив из стандартной библиотеки std::map.

```
class ShapeFactory
{
```

```
public:
   typedef Shape* (*CreateShapeCallback)();
private:
   typedef std::map<int, CreateShapeCallback> CallbackMap;
public:
   // Возвращает значение true, если регистрация прошла успешно
   bool RegisterShape(int ShapeId,
      CreateShapeCallback CreateFn);
   // Возвращает значение true, если фигура ShapeId
   // уже зарегистрирована
   bool UnregisteredShape(int ShapeId);
   Shape* CreateShape(int ShapeId);
private:
   CallbackMap callbacks_;
};
```

Это общая схема масштабируемой фабрики. Фабрика является масштабируемой, поскольку при добавлении нового класса, производного от класса Shape, в код не нужно вносить никаких изменений. Класс ShapeFactory разделяет ответственность: каждая новая фигура должна зарегистрироваться в фабрике, вызвав функцию RegisterShape и передав ей целочисленный идентификатор и указатель на функцию, создающую объект. Обычно эта функция состоит всего из одной строки.

```
Shape*& CreateLine()
{
   return new Line;
};
```

Реализация класса Line также должна зарегистрировать эту функцию в объекте класса ShapeFactory, используемом в приложении. Обычно доступ к этому объекту является глобальным.[2] Регистрация выполняется вместе с инициализацией. Связь класса Line с фабрикой ShapeFactory устанавливается следующим образом.

```
// Модуль реализации класса Line
// Создаем безымянное пространство имен,
// чтобы сделать функцию невидимой из других модулей
namespace
{
   Shape* CreateLine()
   {
      return new Line;
   }
   // Идентификатор класса Line
   const int LINE = 1;
   // Допустим, что фабрика TheShapeFactory —
   // фабрика синглтонов(см. главу 6)
   const bool registered =
      TheShapeFactory::Instance().RegisterShape(
         LINE, CreateLine);
}
```

Благодаря возможностям, предоставленным стандартным ассоциативным массивом std::map, класс ShapeFactory реализуется легко. По существу, функции-члены класса ShapeFactory переадресуют аргументы функции-члену callback_.

---

[2] Это устанавливает связь между фабриками объектов и синглтонами. Действительно, в большинстве случаев фабрики *являются* синглтонами. Ниже в этой главе мы обсудим, как используются фабрики с синглтонами, описанными в главе 6.

```
bool ShapeFactory::RegisterShape(int shapeId,
   CreateShapeCallback createFn)
{
   return callback_.insert(
     CallbackMap::value_type(shapeId, createFn)).second;
}

bool ShapeFactory::UnregisterShape(int shapeId)
{
   return callbacks_.erase(shapeId) == 1;
}
```

Если вы не знакомы с шаблонным классом std::map, предыдущий код нуждается в пояснениях.

- Класс std::map содержит пары, состояшие из ключей и данных. В нашем случае ключами являются целочисленные идентификаторы фигур, а данные состоят из указателя на функцию. Такие пары имеют тип std::pair<const int, CreateShapeCallback>. При вызове функции insert нужно передавать ей объект этого типа. Поскольку это выражение слишком длинное, в классе std::map используется его синоним value_type. В качестве альтернативы можно использовать также тип std::make_pair.
- Функция-член insert возвращает другую пару, на этот раз состоящую из итератора (ссылающегося на только что вставленный элемент) и булевской переменной, принимающей значение true, если значения в ассоциативном массиве до этого момента не было, и false — в противном случае.
- Функция-член erase возвращает количество удаленных элементов.

Функция-член CreateShape просто извлекает указатель на функцию, соответствующий полученному идентификатору типа, и вызывает ее. Если возникает ошибка, генерируется исключительная ситуация.

```
Shape* ShapeFactory::CreateShape(int shapeId)
{
   CallbackMap::const_iterator i = callbacks_.find(shapeId);
   if (i == callbacks_.end())
   {
      // не найден
      throw std::runtime_error("Неизвестный идентификатор");
   }
   // Вызываем функцию для создания объекта
   return (i->second)();
}
```

Посмотрим, что дает нам этот простой класс. Вместо громоздкого оператора switch мы получили динамическую схему, требующую, чтобы каждый тип регистрировался в фабрике. Это распределяет ответственность между классами. Теперь, определяя новый класс, производный от класса Shape, можно просто *добавлять* файлы, а не *модифицировать* их.

8.4. Идентификаторы типов

Осталась одна проблема — управление идентификаторами типов. По-прежнему добавление новых идентификаторов типов требует дисциплинированности и централизованного контроля. Добавляя новый класс, программист обязан проверять все существующие идентификаторы типов, чтобы новый идентификатор не совпал со старыми. Если все же совпадение произошло, второй вызов функции-члена Reg-

isterShape с тем же самым идентификатором не выполняется, и объект этого типа не создается.

Эту проблему можно решить, выбрав для идентификаторов более мощный тип, чем int. В нашем проекте тип int вовсе не требуется. Нужны лишь типы, которые могут быть ключами ассоциативного массива, т.е. типы, поддерживающие операторы == и <. (Вот почему следует применять ассоциативные массивы, а не векторы.) Например, идентификаторы типов можно хранить в виде строк, считая, что каждый класс представлен своим именем: идентификатором типа Line является строка "Line", типа Polygon — строка "Polygon" и т.д. Это минимизирует вероятность совпадения идентификаторов, поскольку имена классов уникальны.

Если вы проводите каникулы, изучая язык С++, предыдущий параграф будет для вас предупредительным сигналом. Давайте попробуем применить класс type_info! Класс std::type_info является частью информации о типах времени выполнения программы (Runtime Type Information — RTTI), предусмотренных языком С++. Ссылку на класс std::type_info можно получить, применив оператор typeid к типу или выражению. Класс std::type_info содержит функцию-член name, возвращающую указатель типа const char*, ссылающийся на имя типа. Указатель, возвращаемый оператором typeid(Line).name(), ссылается на строку "class Line", что и требовалось.

Проблема состоит в том, что не все компиляторы поддерживают этот оператор. Способ, которым реализована функция type_info::name, позволяет применять ее лишь для отладки. Нет никакой гарантии, что данная строка действительно представляет собой имя класса, и, что еще хуже, нет никакой гарантии, что эта строка является единственной в рамках приложения. (Да, пользуясь функцией std::type_info::name, вы можете получить два класса с одним и тем же именем.) И совсем убийственный аргумент: нет гарантии, что имя типа постоянно. Никто не может обещать, что оператор typeid(Line).name() вернет то же самое имя при повторном выполнении программы. Живучесть реализации — важное качество фабрик, а функция std::type_info::name является *неустойчивой*. Итак, хотя класс std::type_info на первый взгляд подходит для создания фабрик, на самом деле он совершенно неприемлем.

Вернемся к вопросу об управлении идентификаторами типов. Генерировать идентификаторы можно с помощью датчика случайных чисел. Этот датчик нужно вызывать каждый раз, когда создается новый объект. При этом новое значение следует запомнить и больше никогда не изменять.[3] Это решение кажется довольно примитивным, однако вероятность того, что за тысячу лет работы датчик выдаст повторяющееся значение, равно 10^{-20}.

Итак, можно сделать единственный вывод: управление идентификаторами типов не входит в компетенцию фабрики объектов. Поскольку язык С++ не может гарантировать уникальность и устойчивость идентификатора типов, управление ими выходит за рамки языка, и ответственность за его реализацию следует возложить на программиста.

---

[3] Фабрика COM-объектов, разработанная компанией Microsoft, использует именно такой подход. Она содержит алгоритм, генерирующий уникальный 128-битовый идентификатор, называемый глобальным уникальным идентификатором (Global Unique Identifier — GUID) COM-объектов. Этот алгоритм основан на уникальности серийного номера сетевой карты или при отсутствии карты даты, времени и других переменных параметров, характеризующих состояние компьютера.

Мы описали все компоненты типичной фабрики объектов и привели прототип реализации. Перейдем теперь к следующему этапу — от частного к общему. Затем, обогатившись знаниями, вернемся к частному.

8.5. Обобщение

Перечислим элементы, которые мы упомянули, обсуждая фабрики объектов. Это даст нам пищу для размышлений при создании обобщенной фабрики объектов.

- *Конкретное изделие* (concrete product). Фабрика производит изделие в виде объекта.
- *Абстрактное изделие* (abstract product). Изделие создается на основе наследования базового типа (в нашем примере — класса Shape). Изделие — это объект, тип которого принадлежит определенной иерархии. Базовый тип этой иерархии представляет собой абстрактное изделие. Фабрика обладает полиморфным поведением, т.е. она возвращает указатель на абстрактное изделие, не передавая знания о типе конкретного изделия.
- *Идентификатор типа изделия* (product type identifier). Это объект, идентифицирующий тип конкретного изделия. Как уже указывалось, идентификатор типа необходим для создания изделия, поскольку система типов языка C++ является статической.
- *Прозводитель изделия* (product creator). Функция или функтор специализируются на создании одного, точно заданного типа объектов. Производитель изделия моделируется с помощью указателя на функцию.

Обобщенная фабрика объединяет в себе все эти элементы для создания точно определенного интерфейса, а также задает по умолчанию параметры, характерные для наиболее широко распространенных ситуаций.

На первый взгляд все перечисленные выше элементы можно преобразовать в шаблонные параметры класса Factory. Однако есть одно исключение: конкретное изделие не обязано быть известным фабрике. Если бы мы должны были точно указывать тип конкретного изделия, то получили бы классы Factory для каждого конкретного изделия отдельно. Это противоречит нашей цели — изолировать класс Factory от конкретных типов. Тип фабрики должен зависеть только от абстрактного изделия.

Итак, подведем промежуточный итог в виде следующего шаблона.

```
template
<
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator
>
clsss Factory
{
public:
   bool Register(const IdentifierType& id, ProductCreator creator)
   {
      return associations_.insert(
         AssocMap::value_type(id, creator)).second;
   }
   bool Unregistered(const IdentifierType& id)
   {
      return associations_.erase(id) == 1;
   }
```

```
      AbstractProduct* CreateObject(const IdentifierType& id)
      {
         typename AssocMap::const_iterator i =
            associations_find(id);
         if (i != associations_.end())
         {
            return(i->second)();
         }
         обработка ошибок
      }
   private:
      typedef std::map<IdentifierType, AbstractProduct>
         AssocMap;
      AssocMap associations_;
   };
```

Осталось только уточнить, что означает "обработка ошибок". Следует ли генерировать исключительную ситуацию, если производитель изделия не зарегистрировался в фабрике? Возвращать в этом случае нулевой указатель? Прекращать работу программы? Динамически загружать ту же самую библиотеку, регистрировать ее на лету и повторять выполнение операции? Выбор зависит от конкретной ситуации. В каждой из них может оказаться разумным одно из перечисленных решений.

Наша обобщенная фабрика должна давать пользователю возможность настраивать ее на любое из перечисленных выше действий и предусматривать разумное поведение по умолчанию. Следовательно, код, предназначенный для обработки ошибок, должен быть выделен из функции-члена `CreateObject` в отдельную стратегию `FactoryError` (глава 1). Эта стратегия состоит только из одной функции `OnUnknownType`, а класс `Factory` предоставляет этой функции шанс (и соответствующую информацию) для принятия любого разумного решения.

Стратегия `FactoryError` очень проста. Она представляет собой шаблонный класс с двумя параметрами: `IdentifierType` и `AbstractProduct`. Если класс `FactoryErrorImpl` представляет собой реализацию стратегии **`FactoryError`**, должно применяться следующее выражение.

```
   FactoryErrorImpl<IdentifierType, AbstractProduct> factoryErrorImpl;
   IdentifierType id;
   AbstractProduct* pProduct = factoryErrorImpl.OnUnknownType(id);
```

Класс `Factory` использует класс `FactoryErrorImpl` в качестве спасательного круга: если класс `CreateObject` не может найти ассоциацию в своем внутреннем ассоциативном массиве, он использует функцию-член `FactoryErrorImpl<IdentifierType, AbstractProduct>::OnUnknownType` для извлечения указателя на абстрактное изделие. Если функция `OnUnknownType` генерирует исключительную ситуацию, она передается за пределы класса `Factory`. В противном случае функция `CreateObject` просто возвращает результат выполнения функции `OnUnknownType`.

Попробуем закодировать эти дополнения и изменения (выделенные в тексте программы полужирным шрифтом).

```
   template
   <
      class AbstractProduct,
      typename IdentifierType,
      typename ProductCreator,
      template<typename, class>
```

```
      class FactoryErrorPolicy
>
class Factory
   : public FactoryErrorPolicy<IdentifierType, AbstractProduct>
{
public:
   AbstractProduct* CreateObject(const IdentifierType& id)
   {
      typename AssocMap::const_iterator i =
         association_.find(id);
      if (i != associations_.end())
      {
         return (i->second)();
      }
      return OnUnknownType(id);
   }
private:
   ... остальные функции и данные остаются неизменными ...
};
```

По умолчанию реализация стратегии FactoryError генерирует исключительную ситуацию. Класс исключительной ситуации должен отличаться ото всех других типов, чтобы клиентский код мог распознать его и принять соответствующие решения. Кроме того, этот класс должен быть производным от одного из стандартных классов исключительных ситуаций, чтобы клиентский код мог перехватывать все виды ошибок в одном блоке catch. Стратегия DefaultFactoryError содержит вложенный класс исключительной ситуации (с именем Exception)[4], производный от класса std::exception.

```
template <class IdentifierType, class ProductType>
class DefaultFactoryError
{
public:
   class Exception : public std::exception
      {
      public:
         Exception(const IdentifierType& unknownId)
            : unknownId_(unknownId)
         {
         }
         virtual const char* what()
         {
            return "Фабрике передается неизвестный тип.";
         }
         const IdentifierType GetId()
         {
            return unknownId_;
         };
      private:
         IdentifierType unknownId_;
      };
protected:
   StaticProductType* OnUnknownType(const IdentifierType& id)
   {
     throw Exception(id);
   }
};
```

[4] Давать этому классу другое имя (например FactoryException) нет никакой необходимости, поскольку этот тип определен внутри шаблонного класса DefaultFactoryError.

Другие, более сложные реализации стратегии FactoryError могут искать идентификатор типа и возвращать указатель на подходящий объект, возвращать нулевой указатель (если генерировать исключительную ситуацию нежелательно), генерировать объект исключительной ситуации или прекращать выполнение программы. Уточнять поведение класса можно с помощью новых реализаций стратегии FactoryError, передавая их в качестве четвертого аргумента класса Factory.

8.6. Мелкие детали

На самом деле реализация класса Factory в библиотеке Loki не использует класс std::map. Вместо него она применяет класс AssocVector, оптимизированный на редкие вставки и часто повторяющиеся операции поиска элементов. Эта ситуация характерна для класса Factory. Класс AssocVector подробно описан в главе 11.

В первоначальном варианте класса Factory ассоциативный массив считался настраиваемым, поскольку был шаблонным параметром. Однако часто класс AssocVector полностью удовлетворяет всем требованиям класса Factory. Кроме того, использование стандартных контейнеров в качестве шаблонных шаблонных параметров не соответствует стандарту языка. Вот почему программисты, занимающиеся реализацией стандартных контейнеров, могут добавлять новые шаблонные аргументы, задавая их значения по умолчанию.

Сосредоточим теперь свое внимание на шаблонном параметре ProductCreator. Во-первых, он должен обладать функциональным поведением (допускать применение оператора () и не иметь аргументов) и возвращать указатель, который можно преобразовывать в тип AbstractProduct*. В конкретной реализации, показанной выше, объект класса ProductCreator был просто указателем на функцию. Этого достаточно, если нужно создавать объекты, вызывая оператор new, как это бывает в большинстве случаев. Следовательно, в качестве типа ProductCreator, задаваемого по умолчанию, можно выбрать следующий:

```
AbstractProduct* (*)()
```

Этот тип выглядит весьма странно, так как не имеет имени. Если после звездочки указать имя, поместив его внутри скобок, тип станет указателем на функцию, не получающую никаких параметров и возвращающую указатель на класс AbstractProduct. Если все это вас по-прежнему удивляет, обратитесь к главе 5, в которой обсуждаются указатели на функцию.

Кстати, в этой главе описан очень интересный шаблонный параметр, который можно передавать классу Factory в качестве параметра ProductCreator, а именно: Functor<AbstractProduct*>. Этот параметр обеспечивает отличную гибкость: можно создавать объекты, вызывая простую функцию, функцию-член или функтор, а также связывать с ними соответствующие параметры. Код, обеспечивающий связь между ними, содержится в классе Functor.

Объявление шаблонного класса Factory выглядит следующим образом.

```
template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

Теперь класс Factory полностью готов.

8.7. Фабрика клонирования

Несмотря на то что генетические фабрики, клонирующие универсальных солдат, — идея страшноватенькая, клонировать объекты языка C++ — занятие совершенно безвредное и даже полезное. Цель клонирования объектов, которую мы рассмотрим, несколько отличается от того, что мы имели в виду раньше: мы больше не должны создавать объекты с самого начала. У нас есть указатель на полиморфный объект, и мы хотели бы создать точную копию этого объекта. Поскольку тип полиморфного объекта точно не известен, мы на самом деле не знаем, какой именно объект создать. В этом и заключается проблема.

Поскольку оригинал у нас под рукой, можно применить классический полиморфизм. Следовательно, при клонировании объекта нужно объявить в базовом классе виртуальную функцию-член `Clone` и заместить ее в производном классе. Рассмотрим клонирование объектов на примере иерархии геометрических фигур.

```
class Shape
{
public:
   virtual Shape* Clone() const = 0;
   ...
};

class Line : public Shape
{
public:
   virtual Line* Clone() const
   {
      return new Line(*this);
   }
   ...
};
```

Функция-член `Line::Clone` не возвращает указатель на класс `Shape`, поскольку мы применили свойства языка C++, называемые *ковариантными типами возвращаемых значений* (covariant return types). Благодаря этому замещенная виртуальная функция может возвращать указатель на производный класс, а не на базовый. Теперь, следуя идиоме, мы должны реализовать аналогичную функцию `Clone` в каждом классе, который добавляется в иерархию. Эти функции имеют одно и то же содержание: создается объект класса `Poligon`, возвращается указатель `new Polygon(*this)` и т.д.

Эта идиома работает, но имеет ряд недостатков.

- Если базовый класс изначально не предназначался для клонирования (в нем не объявлялась виртуальная функция, эквивалентная функции `Clone`) и модификации, то эту идиому применять нельзя. Такая ситуация возникает, например, когда вы пишете приложение, используя в качестве базовых классы из какой-нибудь библиотеки.
- Даже если все классы можно модифицировать, эта идиома требует особой осторожности. Отсутствие реализации функции `Clone` в производных классах компилятор не замечает, что может привести в непредсказуемым последствиям при выполнении программы.

Первый недостаток очевиден, поэтому перейдем к обсуждению второго. Представим себе, что, создавая класс `DottedLine`, производный от класса `Line`, мы забыли заместить функцию `DottedLine::Clone`. Кроме того, допустим, что у нас есть указа-

тель на объект класса Shape, который на самом деле ссылается на объект класса DottedLine, и мы вызываем из этого объекта функцию Clone.

```
Shape* pShape;
...
Shape* pDuplicateShape = pShape->Clone();
```

Вызывается функция Line::Clone, возвращающая объект класса Line. Это очень неприятно, поскольку мы предполагали, что указатель pDuplicateShape имеет тот же динамический тип, что и указатель pShape. Оказывается, это совсем не так, что может вызвать массу проблем — от вывода неожиданных типов до краха приложения.

К сожалению, надежных средств, позволяющих компенсировать второй недостаток, не существует. На языке C++ нельзя сказать: "Я определяю эту функцию и хочу, чтобы она замещалась во всех производных классах". Если вы не хотите неприятностей, придется определить замещаемую функцию Clone в каждом классе.

Если указанную идиому немного усложнить, можно организовать приемлемую проверку типа во время выполнения программы. Сделаем функцию Clone открытой и невиртуальной. Внутри класса она будет вызывать *закрытую виртуальную функцию* DoClone, гарантируя эквивалентность динамических типов. Соответствующий код проще, чем его объяснение.

```
class Shape
{
    ...
public:
    Shape* Clone() const // Невиртуальная функция
    {
        // Переадресовываем вызов функции DoClone
        Shape* pClone = DoClone();
        // Проверяем эквивалентность типов
        // (этот тест может быть сложнее, чем макрос assert)
        assert(typeid(*pClone) == typeid(*this));
        return pClone;
    }
private:
    virtual Shape* DoClone() const = 0; // Закрытая функция
};
```

Единственным недостатком этого подхода является тот факт, что теперь нельзя использовать ковариантные типы возвращаемых значений.

Все наследники класса Shape должны замещать функцию DoClone, оставляя ее закрытой, чтобы клиенты не могли вызвать ее. Функция Clone должна оставаться в стороне. Клиенты используют только функцию Clone, осуществляющую проверку типов во время выполнения программы. Очевидно, что и здесь мы не гарантированы от появления программистских ошибок, например, замещения функции Clone или объявления функции DoClone открытой.

Не забывайте, что если все классы, входящие в иерархию, изменить невозможно (такая иерархия называется *закрытой*), и если они не предназначены для клонирования, эту идиому реализовать все равно не удастся. Такая ситуация встречается довольно часто, поэтому нам следует поискать альтернативу.

На помощь может прийти специальная фабрика объектов. Она свободна от недостатков, указанных выше, но немного снижает производительность программы — вместо виртуального вызова выполняется поиск по карте и вызов через указатель на функцию. Поскольку количество классов приложения на самом деле никогда не бы-

вает очень большим (ведь они создаются людьми, не так ли?), карта обычно невелика, и задержка становится незначительной.

Основная идея состоит в том, что идентификатор типа и изделие имеют одинаковый тип. На вход фабрики поступает дублируемый объект в виде идентификатора типа, а на выход возвращается новый объект, представляющий собой точную копию этого идентификатора. Точнее говоря, их тип не совсем одинаков: тип `IdentifierType` из фабрики клонирования — это *указатель* на объект класса `AbstractProduct`. На самом деле фабрика получает на вход указатель на клонируемый объект, а возвращает — указатель на клон.

А какие ключи хранятся в ассоциативном массиве? Они не могут быть указателями на объекты класса `AbstractProduct`, поскольку количество элементов ассоциативного массива не зависит от количества объектов в программе. Каждый элемент этого массива должен соответствовать отдельному *типу* клонируемого объекта, что вновь приводит нас к стандартному классу `std::type_info`. Идентификатор типа, передаваемый фабрике клонирования при необходимости создать новый объект, отличается от идентификатора типа, хранящегося в ассоциативном массиве. Это не позволяет нам повторно использовать написанный ранее код. Кроме того, производителю изделия теперь нужен указатель на клонируемый объект. В фабрике, которую мы разработали ранее, параметры были не нужны.

Подведем итоги. Фабрика клонирования получает на вход указатель на объект класса `AbstractProduct`. Она применяет к этому объекту оператор `typeid` и получает ссылку на объект класса `std::type_info`. Затем она ищет этот объект в закрытом ассоциативном массиве. (Функция-член `before` класса `std::type_info` устанавливает отношение порядка между объектами этого типа. Это позволяет использовать ассоциативный массив и осуществлять быстрый поиск.) Если элемент не найден, вызывается производитель изделия, которому передается указатель на объект класса `AbstractProduct`, введенный пользователем.

Поскольку у нас уже есть шаблонный класс `Factory`, реализация класса `CloneFactory` упрощается. (Вы можете найти ее в библиотеке Loki.) Класс `CloneFactory` немного отличается от класса `Factory`.

- Класс `CloneFactory` использует класс `TypeInfo`, а не `std::type_info`. Класс `TypeInfo`, рассмотренный в главе 2, представляет собой оболочку указателя на объект класса `std::type_info`. В нем определены инициализация, оператор присваивания, операторы == и <, необходимые для работы с ассоциативным массивом. Первый оператор переадресовывает вызов оператору класса `std::type_info::operator==`, а второй — функции `std::type_info::before`.
- В классе больше нет типа `IdentifierType`, поскольку идентификатор типа является неявным.
- Шаблонный параметр `ProductCreator` по умолчанию задает тип `AbstractProduct*(*)(AbstractProduct*)`.
- Класс `IdToProductMap` теперь заменен классом `AssocVector<TypeInfo, ProductCreator>`.

Класс `CloneFactory` имеет следующий вид.

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(AbstractProduct*),
```

```
    template<typename, class>
       class FactoryErrorPolicy = DefaultFactoreError
>
class CloneFactory
{
public:
   AbstractProduct* CreateObject(const AbstractProduct* model);
   bool Register(const TypeInfo&,
      ProductCreator creator);
   bool Unregister(const TypeInfo&);
private:
   typedef AssocVector<TypeInfo, ProductCreator>
      IdToProductmap;
   IdToProductMap associations_;
};
```

Шаблонный класс CloneFactory представляет собой полное решение задачи клонирования объектов, принадлежащих закрытой иерархии классов (т.е. иерархии, которую невозможно модифицировать). Своей простотой и эффективностью этот класс обязан концепциям, описанным в предыдущих разделах, а также информации о типах, предоставляемой операторами typeid и классом std::type_info. Если бы механизма RTTI не существовало, фабрику клонирования было намного труднее реализовать.

8.8. Использование фабрики объектов в сочетании с другими обобщенными компонентами

В главе 6 был разработан вспомогательный класс SingletonHolder. Поскольку фабрики имеют глобальную природу, естественно использовать класс Factory вместе с классом SingletonHolder. Их легко объединить с помощью оператора typedef.

```
typedef SingletonHolder<Factory<Shape, std::string> > ShapeFactory;
```

Разумеется, для уточнения проектных решений в классы SingletonHolder и Factory можно добавлять аргументы, но все это происходит в одном месте. Следовательно, теперь можно собрать группу важных решений в одном месте и применить класс ShapeFactory. С помощью простого определения типа, указанного выше, можно выбирать способ функционирования фабрики и синглтона. Единственная строка кода отдает компилятору приказание сгенерировать соответствующий код, аналогично тому, как вызов функции с разными аргументами определяет разные пути ее выполнения. Поскольку в нашем случае все это происходит во время компиляции, основной упор переносится на проектные решения, а не на выполнение программы. Разумеется, это не может не влиять на выполнение программы, однако это влияние имеет косвенный характер. Когда вы пишете “обычный” код, вы определяете, что будет происходить во время выполнения программы. Когда вы пишете определение типа, подобное приведенному выше, вы указываете, что должно произойти во время компиляции программы — это можно назвать вызовом функций, генерирующих код.

Как указывалось в начале этой главы, большой интерес вызывает комбинация класса Factory с классом Functor.

```
typedef SingletonHolder
<
   Factory
   <
      Shape, std::string, Functor<Shape*>
   >
>
ShapeFactory;
```

Использование класса Functor обеспечивает большую гибкость при создании объектов. Теперь новые объекты класса Shape и его потомков можно создавать всевозможными способами, регистрируя в фабрике разные функторы.

8.9. Резюме

Фабрики объектов — важный компонент программ, использующих полиморфизм. Они позволяют создавать объекты, когда их тип либо совсем недоступен, либо несовместим с конструкциями языка.

В основном фабрики объектов применяются в объектно-ориентированных приложениях и библиотеках, а также в различных схемах управления живучестью объектов и потоками. Схема управления потоками детально проанализирована на конкретном примере. По существу, она сводится к распределению типов между разными файлами реализации, обеспечивая высокую модульность программы. Несмотря на то что фабрика остается основным средством создания объектов, она не обязана собирать информацию обо всех статических типах, входящих в иерархию. Вместо этого, она заставляет каждый тип зарегистрироваться на фабрике. В этом заключается принципиальная разница между "правильным" и "неправильным" подходом.

Информацию о типе во время выполнения программы на языке C++ передавать трудно. Эта особенность носит принципиальный характер для всего семейства языков, которому принадлежит язык C++, поэтому приходится применять вместо типа его идентификатор. Идентификаторы типов тесно связаны с вызываемыми сущностями, создающими объекты (глава 5). Для определения этих ограничений реализована конкретная фабрика объектов, обобщенная в виде шаблонного класса.

В заключении рассмотрены фабрики клонирования, позволяющие создавать дубликаты полиморфных объектов.

8.10. Краткий обзор шаблонного класса Factory

- Объявление класса Factory имеет следующий вид.

  ```
  template
  <
     class AbstractProduct,
     class IdentifierType,
     class ProductCreator = AbstractProduct* (*)(),
     template<typename, class>
        class FactoryErrorPolicy = DefaultFactoryError
  >
  class Factory;
  ```

- Класс AbstractProduct является базовым классом иерархии, для которой создается фабрика объектов.
- Класс IdentifierType представляет тип, входящий в иерархию. Он должен иметь отношение порядка, позволяющее хранить его в объекте класса std::map. Обычно в качестве идентификаторов типа используются строки и целые числа.
- Класс ProductCreator представляет собой вызываемую сущность, создающую объект. Этот класс должен поддерживать оператор (), не иметь параметров и возвращать указатель на объекты класса AbstractProduct. Объект класса ProductCreator всегда регистрируется вместе с идентификатором типа.

- В классе `Factory` реализуются следующие примитивы.

```
bool Register(const IdentifierType&, ProductCreator creator);
```

Эта функция регистрирует производитель изделия вместе с идентификатором типа. Если регистрация прошла успешно, она возвращает значение `true`, в противном случае возвращается значение `false` (если производитель изделия с тем же идентификатором типа уже зарегистрирован ранее).

```
bool Ungeristered(const IdentifierType& id);
```

Эта функция аннулирует регистрацию идентификатора заданного типа. Если этот идентификатор был зарегистрирован ранее, функция возвращает значение `true`.

```
AbstractProduct* CreateObject(const IdentifierType& id);
```

Эта функция выполняет поиск идентификатора типа во внутреннем ассоциативном массиве. Если идентификатор найден, она вызывает соответствующий производитель объектов данного типа и возвращает результат его работы. Если идентификатор не найден, возвращается результат работы функции `FactoryErrorPolicy<IdentifierType, AbstractProduct>::OnUnknownType`. По умолчанию реализация класса `FactoryErrorPolicy` генерирует исключительную ситуацию вложенного типа `Exception`.

8.11. Краткий обзор шаблонного класса CloneFactory

- Объявление класса `CloneFactory` имеет следующий вид.

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(const AbstractProduct*),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory;
```

- Класс `AbstractProduct` является базовым классом иерархии, для которой создается фабрика клонирования.
- Класс `ProductCreator` представляет собой вызываемую сущность, создающую дубликат объекта, передаваемого как параметр, и возвращающую указатель на клон.
- В классе `CloneFactory` реализуются следующие примитивы.

```
bool Register(const TypeInfo&, ProductCreator creator);
```

Эта функция регистрирует производитель изделия вместе с типом `TypeInfo` (что позволяет неявно вызывать конструктор копирования класса `std::type_info`). Если регистрация прошла успешно, она возвращает значение `true`, в противном случае возвращается значение `false`.

```
bool Ungeristered(const TypeInfo& typeinfo);
```

Эта функция аннулирует регистрацию производителя объектов заданного типа. Если этот тип был зарегистрирован ранее, функция возвращает значение `true`.

```
AbstractProduct* CreateObject(const AbstractProduct* model);
```

Эта функция выполняет поиск динамического типа объекта `model` во внутреннем ассоциативном массиве. Обнаружив тип, она вызывает соответствующий производитель объектов данного типа и возвращает результат его работы. Если тип не найден, возвращается результат работы функции `FactoryErrorPolicy<OrderedTypeInfo, AbstractProduct>::OnUnknownType`.

9

Шаблон Abstract Factory

В этой главе обсуждается обобщенная реализация шаблона проектирования Abstract Factory (Gamma et al., 1995). Абстрактная фабрика — это интерфейс для создания семейства связанных или взаимозависимых объектов.

Абстрактные фабрики могут быть важными архитектурными компонентами, поскольку они гарантируют, что в системе создаются правильные конкретные объекты. Если вы не хотите, чтобы кнопка FunkyButton появлялась в диалоговом окне ConventionalDialog, используйте шаблон проектирования Abstract Factory, гарантирующий, что эта кнопка появляется только в окне FunkyDialog. Для этого нужно проконтролировать лишь небольшую часть кода, а остальная часть приложения будет работать с абстрактными типами Dialog и Button.

В главе рассматриваются следующие вопросы.

- Область применения шаблона проектирования Abstract Factory.
- Определение и реализация компонентов шаблона Abstract Factory.
- Использование обобщенных функциональных возможностей шаблона Abstract Factory, предоставленных библиотекой Loki, и их расширение.

9.1. Архитектурная роль шаблона Abstract Factory

Представьте себе, что вы разрабатываете компьютерную игру "найти и уничтожить" наподобие Doom или Quake.

Вы хотите соблазнить потенциального покупателя своей игрой, поэтому предусматриваете легкий уровень. На этом уровне вражеские солдаты абсолютно тупые, монстры представляют собой желеобразную массу, а супермонстры настроены крайне дружелюбно.

Более крутых игроков вы соблазняете уровнем повышенной сложности. На этом уровне вражеские солдаты совершают три выстрела в секунду и владеют приемами карате, монстры коварны и вероломны, а супермонстры встречаются постоянно.

Возможная модель этого ужасного мира, населенного врагами и монстрами, может содержать базовый класс Enemy и производные от него интерфейсы Soldier, Monster и SuperMonster. Затем на основе этих интерфейсов для реализации легкого уровня игры создаются *производные классы SillySoldier, SillyMonster и SillySuperMonster*. В заключение для уровня повышенной сложности реализуются классы BadSoldier, BadMonster и BadSuperMonster. В результате возникает иерархия классов, представленная на рис. 9.1.

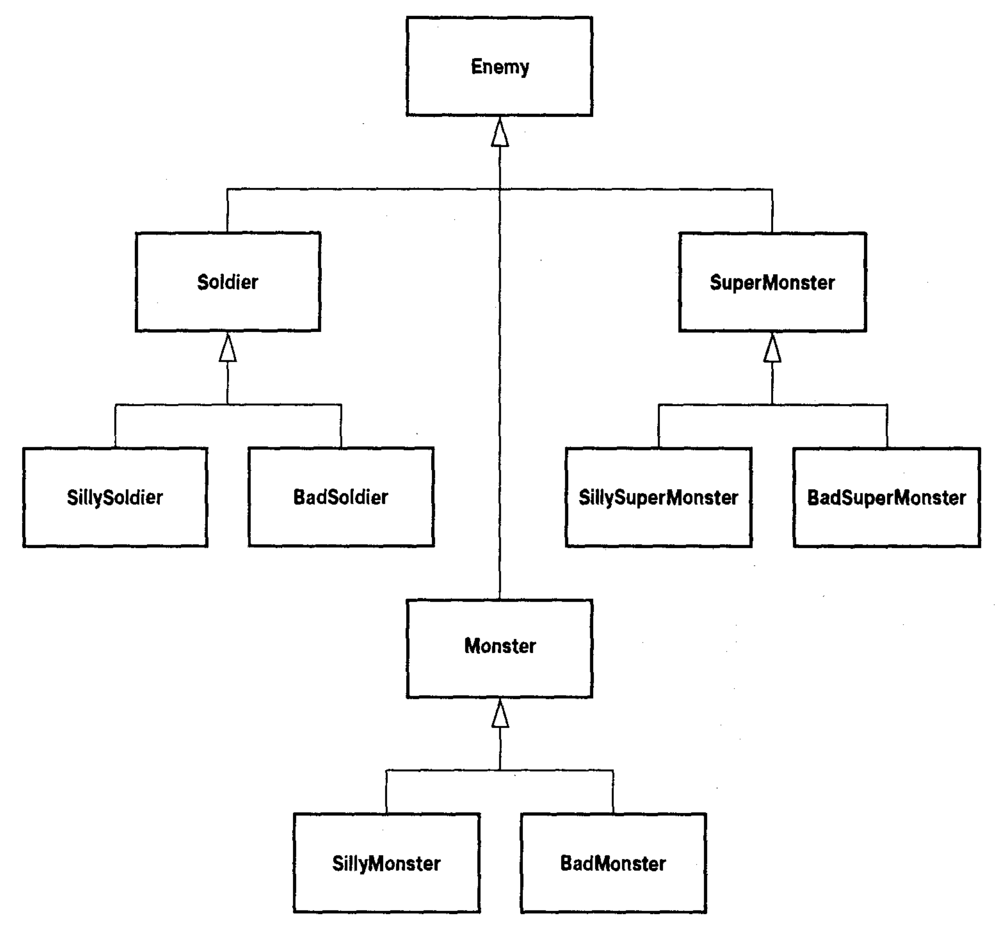


Рис. 9.1. Иерархия классов для игры с двумя уровнями сложности

Стоит напомнить, что в вашей игре конкретизации классов `BadSoldier` и `SillyMonster` никогда не "встречаются" друг с другом. Это не имело бы смысла. Игрок на легком уровне имеет дело с классами `SillySoldier`, `SillyMonster` и `SillySuperonster`, а на уровне повышенной сложности — с классами `BadSoldier`, `BadMonster` и `BadSuperMonster`.

Две категории типов формируют два семейства. Во время игры всегда используются объекты лишь одного конкретного семейства, а комбинации объектов, принадлежащих разным семействам, никогда не возникают.

Было бы прекрасно, если бы это соответствие выполнялось автоматически. В противном случае, если программист не проявил необходимой осторожности, новичок, развлекающийся с объектами класса `SillySoldier`, может неожиданно встретиться в темном углу с объектом класса `BadMonster`. В этом случае обиженные игроки могут потребовать от вас возврата денег.

Поскольку лучше проявлять осторожность заранее, функции, предназначенные для создания объектов, задействованных в игре, следует объединить в одном интерфейсе.

```
class AbstractEnemyFactory
{
public:
   virtual Soldier* MakeSoldier() = 0;
   virtual Monster* MakeMonster() = 0;
   virtual SuperMonster* MakeSuperMonster() = 0;
};
```

Тогда для каждого уровня сложности создается конкретная фабрика врагов, создающая объекты в соответствии с заданной стратегией игры.

```
  class EasyLevelEnemyFactory : public AbstractEnemyFactory
{
public:
   Soldier* MakeSoldier;
   { return new SillySoldier; }
   Monster* MakeMonster()
   { return new SillyMonster; }
   SuperMonster* MakeSuperMonster()
   { return new SillySuperMonster; }
};

class DieHardLevelEnemyFactory : public AbstractEnemyFactory
{
public:
   Soldier* MakeSoldier()
   { return new BadSoldier; }
   Monster* makeMonster()
   { return new BadMonster; }
   SuperMonster* MakeSuperMonster()
   { return new BadSuperMonster; }
};
```

В заключение мы инициализируем указатель на объект класса AbstractEnemyFactory соответствующего конкретного класса.

```
class GameApp
{
   ...
   void SelectLevel()
   {
      if (пользователь выбрал легкий уровень игры)
      {
         pFactory_ = new EasyLevelEnemyFactory;
      }
      else
      {
         pFactory_ = new DieHardLevelEnemyFactory;
      }
   }
private:
   // Используем указатель pFactory_ для создания врагов
   AbstractEnemyFactory* pFactory_;
};
```

Преимущества этого подхода заключаются в том, что он сохраняет все детали создания и сопоставления врагов внутри двух реализаций класса AbstractFactory. Поскольку приложение использует указатель pFactory_ исключительно для ссылки на объект производителя, соответствие объектов уровню игры обеспечивается автоматически.

Шаблон проектирования Abstract Factory предписывает, чтобы все функции, предназначенные для создания объектов, были сосредоточены в отдельном интерфей-

се. Затем программист должен выполнить реализацию этого интерфейса отдельно для каждого семейства создаваемых объектов.

Типы, анонсированные абстрактным фабричным интерфейсом (Soldier, Monster и SuperMonster), называются *абстрактными изделиями* (abstract product). Типы, которые фактически создаются (SillySoldier, BadSoldier, SillyMonster и т.д.), называются *конкретными изделиями* (concrete product). Эти термины уже встречались в главе 8.

Основной недостаток шаблона Abstract Factory заключается в том, что базовый класс фабрики (в нашем примере AbstractEnemyFactory) должен владеть информацией о каждом создаваемом абстрактном изделии. Кроме того, по крайней мере в только что описанной реализации, каждый конкретный фабричный класс зависит от конкретного изделия, которое он создает.

С помощью приемов, описанных в главе 8, можно понизить степень этой зависимости. В той главе мы создавали конкретные объекты, не зная ничего об их типах. Единственным доступным источником информации были идентификаторы типов (например, целые числа или строки).

Однако, чем больше мы снижаем степень зависимости между классами, тем меньше становится объем информации о типах, следовательно, значительно снижается степень типовой безопасности приложения. Это еще один пример классической дилеммы, возникающей в языке C++: высокая степень типовой безопасности или слабая взаимосвязь между классами.

Как это часто бывает, правильное решение можно найти с помощью компромисса. В каждом конкретном приложении следует искать свою золотую середину. Общее правило таково: используйте статическую модель, если можете, и динамическую модель, если вынуждены.

Обобщенная реализация шаблона Abstract Factory, описанная в следующих разделах, обладает интересным свойством, позволяющим снизить степень статической зависимости между классами без снижения степени типовой безопасности.

9.2. Обобщенный интерфейс шаблона Abstract Factory

Как указывалось в главе 3, функциональные возможности класса Typelist позволяют реализовать обобщенный шаблон Abstract Factory. В этом разделе показано, как определяется обобщенный интерфейс AbstractFactory с помощью списков типов.

Пример, описанный в предыдущем разделе, представляет собой типичное применение шаблона проектирования Abstract Factory. Его использование сводится к следующему.

- Определяется абстрактный класс (класс абстрактной фабрики), содержащий одну чисто виртуальную функцию для каждого типа изделий. Виртуальная функция, соответствующая типу T, обычно возвращает указатель типа T*. Она называется CreateT, MakeT или как-то еще.
- Определяется одна или несколько конкретных фабрик, реализующих интерфейс, определенный абстрактной фабрикой. Реализуется каждая из функций-членов, создающих новый объект производного типа с помощью оператора new.

Все это выглядит довольно просто, но при возрастании количества изделий, создаваемых абстрактной фабрикой, программа становится все более ненадежной. Более того, в любой момент программист может внедрить в программу новую реализацию, которая использует другой механизм распределения памяти, или прототип объекта.

В этой ситуации на помощь приходит шаблон Abstract Factory при условии, что он достаточно гибок и позволяет легко реализовать новый механизм распределения памяти и передачи аргументов конструктору.

Напомним, что шаблонный класс GenScatterHierarchy, описанный в главе 3, конкретизирует базовый шаблон, предоставленный пользователем, для каждого типа, приведенного в списке типов. По своей структуре полученная в результате конкретизация класса GenScatterHierarchy наследует все конкретизации шаблона, предоставленного пользователем. Иными словами, если есть шаблон Unit и список типов TList, то класс GenScatterHierarchy<TList, Unit> является наследником класса Unit<T> для каждого типа T, указанного в списке TList.

Класс GenScatterHierarchy может оказаться очень полезным для определения интерфейса абстрактной фабрики — программисту достаточно определить интерфейс, который способен создавать объекты определенного типа, а затем применить этот интерфейс к нескольким типам с помощью класса GenScatterHierarchy.

Интерфейс, способный создавать объекты обобщенного класса T, выглядит следующим образом.

```
template <class T>
class AbstractFactoryUnit
{
public:
   virtual T* DoCreate(Type2Type<T> = 0);
   virtual ~AbstractFactoryUnit() {}
};
```

Этот небольшой шаблон выглядит вполне прилично — виртуальный конструктор и ничего больше[1], — но зачем здесь класс Type2Type? Как указывалось в главе 2, класс Type2Type — это простой шаблон, единственное предназначение которого — обеспечить однозначный выбор перегруженных функций. Ладно, но где тут перегруженные функции? Ведь в классе AbstractFactoryUnit определяется только *одна* функция DoCreate. Однако одной и той же иерархии классов могут принадлежать разные конкретизации класса AbstractFactoryUnit. Класс Type2Type<T> позволяет избежать неоднозначности выбора разных вариантов перегруженной функции DoCreate.

Обобщенный интерфейс AbstractFactory использует класс GenScatterHierarchy в сочетании с классом AbstractFactoryUnit.

```
template
<
   class TList,
   template <class> class Unit = AbstractFactoryUnit
>
class AbstractFactory : public GenScatterHierarchy<TList, Unit>
{
public:
   typedef TList ProductList;
   template <class T> T* Create()
   {
      Unit <T>& unit = *this;
      return unit.DoCreate(Type2Type<T>());
   }
};
```

[1] Виртуальные конструкторы подробно обсуждались в главе 4.

Вот тут-то на арене появляется класс Type2Type, уточняющий, что функция DoCreate вызывается функцией Create. Проанализируем следующий фрагмент кода.

```
// Код приложения
typedef AbstractFactory
<
   TYPELIST_3(Soldier, Monster, SuperMonster)
>
AbstractEnemyFactory;
```

В главе 3 было показано, что шаблонный класс AbstractFactory генерирует иерархию классов, показанную на рис. 9.2. Класс AbstractEnemyFactory является потомком классов AbstractFactoryUnit<Soldier>, AbstractFactoryUnit<Monster> и AbstractEnemyUnit<SuperMonster>. В каждом из них определяется чисто виртуальная функция-член Create, так что класс AbstractEnemyFactory обладает тремя перегруженными функциями Create. Короче говоря, класс AbstractEnemyFactory практически эквивалентен абстрактному классу с тем же именем, определенному в предыдущем разделе.

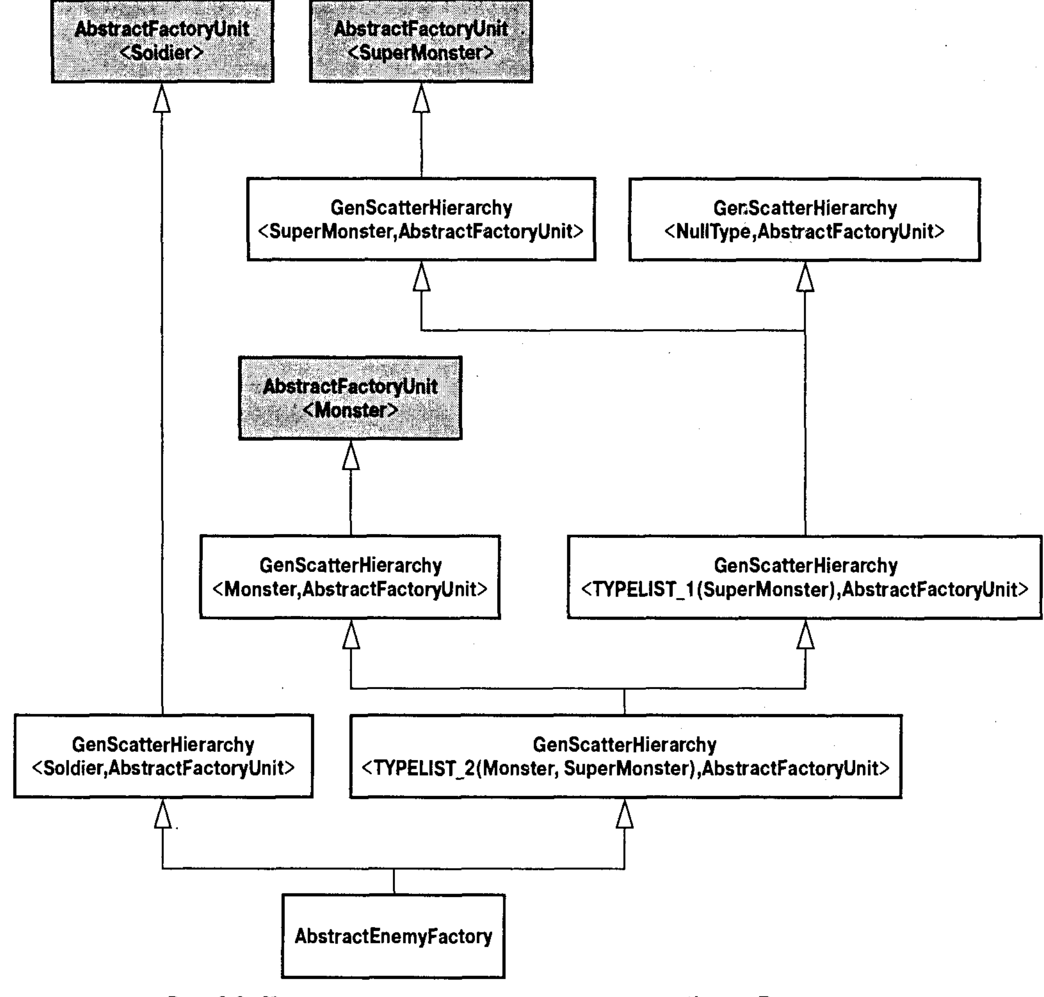


Рис. 9.2. Иерархия классов, генерируемая классом AbstractFactory

Шаблонная функция-член Create из класса AbstractFactory играет роль диспетчера, направляющего запрос на создание объекта соответствующему базовому классу.

```
AbstractEnemyFactory* p = ...;
Monster* pOgre = p->Create<Monster>();
```

Автоматически генерируемая версия класса обладает одним важным преимуществом — класс AbstractEnemyFactory имеет высокий уровень *модульности* (granularity). Программист может автоматически конвертировать ссылку на объект класса AbstractEnemyFactory в ссылку на объект класса AbstractFactory<Soldier>, AbstractFactory<Monster> или AbstractFactory<SuperMonster>. Таким образом, различным частям приложения можно передавать лишь небольшие разделы фабрики. Например, допустим, что некий модуль (Surprises.cpp) должен создавать только объекты класса SuperMonster. Этому модулю можно передавать только указатели или ссылки на объекты класса AbstractFactory<SuperMonster>, так что модуль Surprises.cpp не связан ни с классом Soldier, ни с классом Monster.

Используя модульность класса AbstractFactory, можно сократить количество связей, задействованных в шаблоне проектирования Abstract Factory. Это не снизит степень безопасности интерфейса абстрактной фабрики, поскольку модульным является только интерфейс, а не реализация класса.

Итак, перед нами встает проблема реализации интерфейса. Второе важное преимущество автоматически сгенерированной версии класса AbstractEnemyFactory заключается в том, что его реализация также генерируется автоматически.

9.3. Реализация класса AbstractFactory

Определив интерфейс, перейдем к реализации класса, стремясь сделать это как можно проще.

Поскольку интерфейс использует списки типов, естественно построить обобщенную реализацию класса AbstractFactory с помощью списков типов конкретных изделий. С практической точки зрения реализация класса, соответствующего легкому уровню игры, должна выглядеть следующим образом.

```
// Код приложения
typedef ConcreteFactory
<
   // Абстрактная фабрика, подлежащая реализации
   AbstractEnemyFactory,
   // Стратегия создания объектов
   // (например, с помощью оператора new)
   OpNewFactoryUnit,
   // Конкретные классы, создаваемые данной фабрикой
   TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
>
EasyLevelEnemyFactory;
```

Три аргумента гипотетического (пока) шаблонного класса ConcreteFactory содержат достаточно информации, чтобы полностью реализовать фабрику.

- Класс AbstractEnemyFactory описывает интерфейс абстрактной фабрики, подлежащей реализации, и неявно задает список изделий.
- Класс OpNewFactoryUnit представляет собой стратегию, определяющую, каким образом осуществляется фактическое создание объектов.

- Список типов содержит набор классов, создаваемых фабрикой. Каждый конкретный тип, указанный в списке, соответствует абстрактному типу с тем же индексом в списке типов класса AbstractFactory. Например, класс SillyMonster (индекс 1) представляет собой конкретную разновидность класса Monster, имеющую тот же индекс в определении класса AbstractEnemyFactory.

Описав класс ConsreteFactory, попробуем его реализовать. Немного поразмыслив, легко прийти к выводу, что количество замещений чисто виртуальных функций должно совпадать с количеством определений класса. (В противном случае мы не смогли бы конкретизировать класс ConcreteFactory.) Следовательно, класс ConcreteFactory должен быть производным от класса OpNewFactoryUnit, отвечающего за реализацию функции DoCreate.

Здесь большую помощь может оказать шаблонный класс GenlinearHierarchy (глава 3), поскольку в нем предусмотрены все детали генерации нужных нам реализаций.

Класс AbstractEnemyFactory должен быть корнем иерархии. Все реализации функции DoCreate и окончательный класс EasyLevelEnemyFactory должны быть производными от него. В каждой реализации класса OpNewFactoryUnit должна замещаться одна из трех чисто виртуальных функций DoCreate, определенных в классе AbstractEnemyFactory.

Определим класс OpNewFactoryUnit. Очевидно, класс OpNewFactoryUnit — это шаблонный класс, получающий тип создаваемого объекта в качестве шаблонного параметра. Кроме того, класс GenLinearHierarchy требует, чтобы класс OpNewFactoryUnit получал дополнительный шаблонный параметр и был его потомком. (Класс GenLinearHierarchy использует второй аргумент для генерации линейной (string-shaped) иерархии наследования, изображенной на рис. 3.6.)

```
template <class ConcreteProduct, class Base>
class OpNewFactoryUnit : public Base
{
   typedef typename Base::ProductList BaseProductList;
protected:
   typedef typename BaseProductList::Tail ProductList;
public:
   typedef typename BaseProductList::Head AbstractProduct;
   ConcreteProduct* DoCreate(Type2Type<AbstractProduct>)
   {
      return new ConcreteProduct;
   }
};
```

Чтобы определить, какое абстрактное изделие следует создать, класс OpNewFactoryUnit должен выполнять только одно вычисление типа.

Каждая конкретизация класса OpNewFactoryUnit является компонентом некоей "пищевой" цепи. Каждая конкретизация класса OpNewFactoryUnit "откусывает" голову списка изделий, замещая соответствующую функцию DoCreate и передавая "безголовый" список ProductList вниз по иерархии классов. Таким образом, конкретизация класса OpNewFactoryUnit, находящаяся на вершине иерархии (сразу за классом AbstractEnemyFactory), реализует функцию DoCreate(Type2Type<Soldier>), а конкретизация, находящаяся на дне иерархии, реализует функцию DoCreate(Type2Type<SuperMonster>).

Итак, попробуем разобраться, почему класс OpNewFactoryUnit занимает такое высокое положение в иерархии. Во-первых, класс OpNewFactoryUnit импортирует тип

ProductList из базового класса и присваивает ему новое имя BaseProductList. (Просмотрев определение класса AbstractFactory, легко понять, что он на самом деле экспортирует тип ProductList.) Абстрактное изделие, реализованное классом OpNewFactoryUnit, представляет собой голову списка BaseProductList в соответствии с определением класса AbstractProduct. В заключение класс OpNewFactoryUnit реэкспортирует класс BaseProductList::Tail в качестве типа ProductList. Оставшаяся часть списка передается вниз по иерархии.

Обратите внимание на то, что функция OpNewFactoryUnit::DoCreate не возвращает указатель на объект класса AbstractProduct, как это делает ее настоящий прототип. Вместо этого она возвращает указатель на объект класса ConcreteProduct. Можно ли это по-прежнему квалифицировать как реализацию чисто виртуальной функции? Да, благодаря *ковариантным типам возвращаемых значений* (covariant return types). Язык C++ позволяет возвращать указатель на объект *производного* класса. В этом есть глубокий смысл. Благодаря этим типам программист либо знает точный тип конкретной фабрики, получая максимум информации, либо знает только ее базовый тип, получая меньший объем информации.

Класс ConcreteFactory должен генерировать иерархию, используя класс GenLinearHierarchy. Его реализация вполне очевидна.

```
template
<
   class AbstractFact,
   template <class, class> class Creator = OPNewFactoryUnit;
   class TList = typename AbstractFact::ProductList
>
class ConcreteFactory
   : public GenLinearHierarchy<
     typename TL:Reverse<TList>::Result, Creator, AbstractFact>
{
   typedef typename AbstractFact::ProductList ProductList;
   typedef TList ConcreteProductList;
};
```

Иерархия классов, генерируемая классом GenLinearHierarchy для класса ConcreteFactory, показана на рис. 9.3.

Здесь скрывается одна маленькая хитрость: класс ConcreteFactory должен переворачивать список конкретных изделий, передавая его классу GenLinearHierarchy. Зачем? Для этого нужно вернуться к рис. 3.6, на котором показано, как класс GenLinearHierarchy генерирует иерархию. Этот класс передает типы из списка типов в шаблонный аргумент Unit снизу вверх. Первый элемент списка типов передается конкретизации класса Unit, находящейся на дне иерархии классов. Однако класс OpNewFactoryUnit реализует перегрузку функции DoCreate сверху вниз. Следовательно, класс ConcreteFactory должен переворачивать список TList, используя статический алгоритм TL::Reverse (глава 3), и только после этого передавать его классу GenLinearHierarchy.

Если вы все еще считаете классы AbstractFactory и ConcreteFactory сложными и запутанными, потерпите. Так кажется потому, что классы небрежно обращаются со списками типов. Списки типов представляют собой новое понятие, и, чтобы привыкнуть к нему, требуется время. Если вы будете считать списки типов подобием "черного ящика" — "списки типов по отношению к типам играют ту же роль, что и обычные списки по отношению к значениям", — реализация классов сразу станет

понятнее. Если уж вы *на самом деле* решили использовать списки типов, то ваши возможности станут поистине неограниченными. Не верите? Читайте дальше.

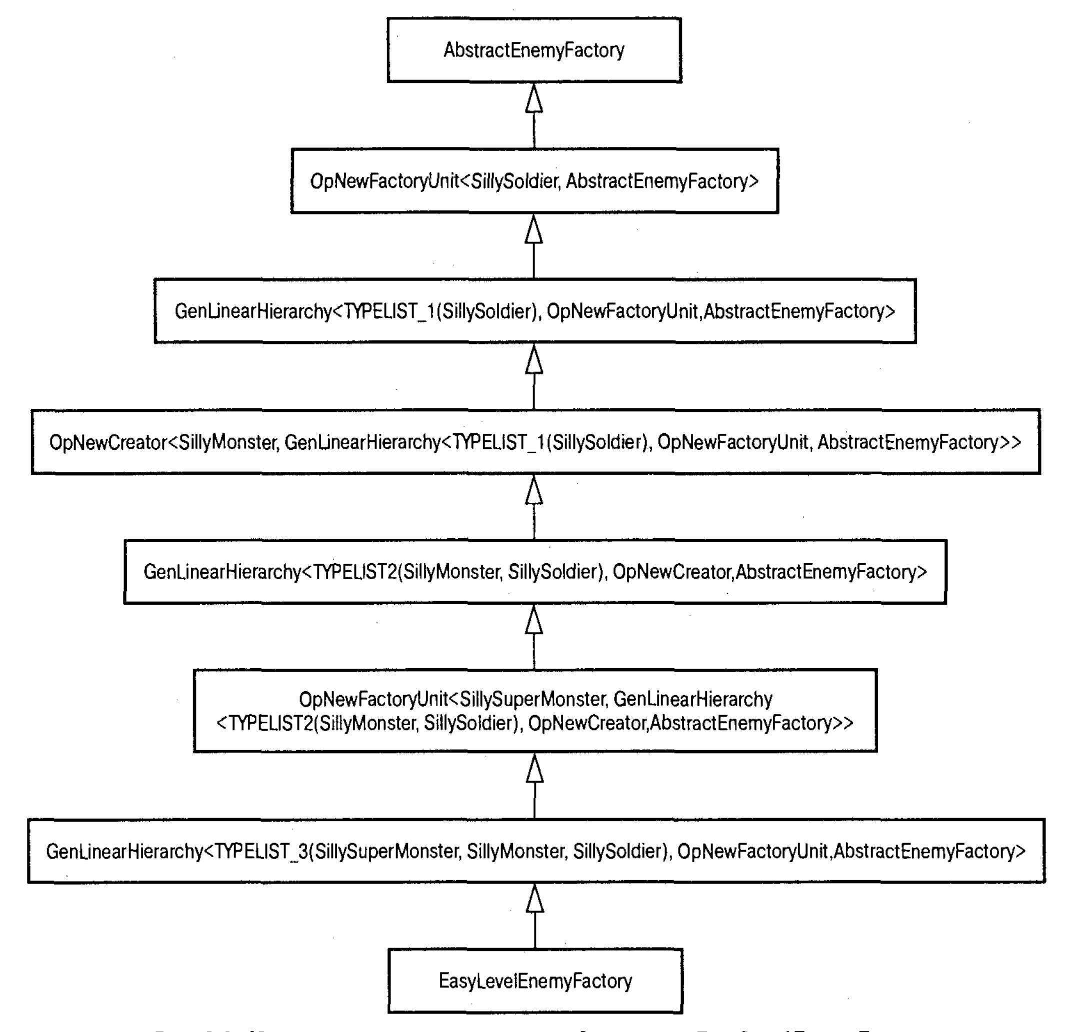


Рис. 9.3. Иерархия классов, генерируемая для класса EasyLevelEnemyFactory

9.4. Реализация шаблона Abstract Factory на основе прототипов

Шаблон проектирования `Prototype` (Gamma et al., 1995) описывает метод создания объектов, начиная с *прототипа*, представляющего собой некий архетип. Новые объекты получаются путем клонирования прототипа. Суть этого способа заключается в том, что функция клонирования является виртуальной.

В главе 8 детально обсуждалась важная проблема, возникающая при создании полиморфных объектов. Эта проблема называется *дилеммой виртуального конструктора*: при создании объекта с нуля нужно знать тип создаваемого объекта, хотя полиморфизм предполагает *отсутствие* точной информации о типе.

Шаблон проектирования Prototype избегает этой дилеммы, используя прототип объекта. Имея прототип, мы можем использовать преимущества виртуальных функций. Дилемма виртуального конструктора по отношению к самому прототипу остается нерешенной, но теперь она носит намного более локальный характер.

В описанном выше примере подход, основанный на применении прототипов при создании врагов, задействованных в компьютерной игре, вынуждает использовать указатели на объекты базовых классов Soldier, Monster и SuperMonster. В этом случае код может выглядеть примерно так.[2]

```
class GameApp
{
    ...
    void SelectLevel()
    {
        if (пользователь выбрал уровень повышенной сложности)
        {
            protoSoldier_.reset(new BadSoldier);
            protoMonster_.reset(new BadMonster);
            protoSuperMonster_.reset(new BadSuperMonster);
        }
        else
        {
            protoSoldier_.reset(new SillySoldier);
            protoMonster_.reset(new SillyMosnter);
            protoSuperMonster_.reset(new SillySuperMosnter);
        }
    }
    Soldier* MakeSoldier()
    {
        // В каждом вражеском классе определяется
        // виртуальная функция Clone
        return pProtoSoldier_->Clone();
    }
    ... Классы MakeMonster и MakeSuperMonster определяются аналогично
...
private:
    // Используем эти прототипы для создания врагов
    auto_ptr<Soldier> protoSoldier_;
    auto_ptr<Monster> protoMonster_;
    auto_ptr<SuperMonster> protoSuperMonster_;
}
```

Разумеется, в реальном коде лучше было бы разделить интерфейс и реализацию. Основная идея заключается в том, чтобы класс GameApp хранил указатели на базовые вражеские классы — прототипы. Класс GameApp использует эти прототипы для создания вражеских объектов, применяя к ним виртуальную функцию Clone.

Реализация шаблона Abstract Factory, основанная на применении прототипов, может хранить указатель на каждый тип изделия и использовать для создания новых изделий функцию Clone.

В классе ConcreteFactory, использующем прототип, больше нет необходимости предусматривать конкретные типы. В нашем примере для создания объектов классов SillySoldier и BadSoldier нужно лишь передать фабрике соответствующие прототипы. Статический тип прототипа — базовый класс Soldier. Фабрика не обязана ни-

---

[2] Учтите, этот код неверно обрабатывает исключительные ситуации. Исправьте его самостоятельно.

чего знать о конкретных типах объектов. Она просто вызывает виртуальную функцию-член Clone из соответствующего прототипа. Это ослабляет зависимость конкретных фабрик от конкретных типов.

Однако, для того чтобы механизм класса GenLinearHierarchy работал правильно, нужен список типов. Напомним, как выглядит объявление класса ConcreteFactory.

```
template
<
   class AbstractFact,
   template <class, class> class Creator,
   class TList
>
class ConcreteFactory;
```

Класс TList — это список конкретных изделий. В классе EasylevelEnemyFactory класс TList был известен под именем
TYPELIST_3 (SillySoldier, SillyMonster, SillySuperMonster).
При использовании шаблона проектирования Pattern класс TList становится неуместным. Однако класс TList по-прежнему необходим классу GenLinearHierarchy для генерации отдельного класса для каждого изделия, указанного в списке абстрактных изделий. Что делать?

В этой ситуации естественно передать *список абстрактных изделий* классу ConcreteFactory в качестве аргумента TList. Теперь класс GenLinearHierarchy сможет генерировать правильное количество классов, а изменять реализацию класса ConcreteFactory не понадобится.

Объявление класса ConcreteFactory принимает следующий вид.

```
template
<
   class AbstractFact,
   template <class, class> class Creator,
   class TList = typename AbstractFact::ProductList
>
class ConcreteFactory;
```

Напомним, что определение класса AbstractFact, приведенное в разделе 9.3, содержит внутренний класс ProductList.

Перейдем теперь к реализации шаблонного класса PrototypeFactoryUnit, содержащего прототип и вызывающего функцию Clone. Его реализация намного проще, чем реализация класса OpNewFactoryUnit, поскольку вместо *двух* списков типов класс PrototypeFactoryUnit работает только с одним списком абстрактных изделий.

```
template <class ConcreteProduct, class Base>
class PrototypeFactoryUnit : public Base
{
   typedef typename Base::ProductList BaseProductlist;
protected:
   typedef typename Base::ProductList TailProductlist;
public:
   typedef typename Base::ProductList::Head AbstractProduct;
   PrototypeFactoryUnit(AbstractProduct* p = 0);
      : pPrototype_(p)
   {}
   friend void DoGetPrototype(const PrototypeFactoryUnit& me,
      AbstractProduct*& pPrototype)
   {
      pPrototype = me.pPrototype_;
```

```
    }
    friend void DoSetPrototype(PrototypeFactoryUnit& me,
        AbstractProduct* pObj)
    {
        me.pPrototype_=pObj;
    }
    template <class U>
    void GetPrototype(AbstractProduct*& p)
    {
        return DoGetPrototype(*this, p);
    }
    template <class U>
    void SetPrototype(U* pObj)
    {
        DoSetPrototype(*this, pObj);
    }
    AbstractProduct* DoCreate(Type2Type<AbstractProduct>)
    {
        assert(pPrototype_);
        return pPrototype_->Clone();
    }
private:
    AbstractProduct* pPrototype_;
};
```

Шаблонный класс `PrototypeFactoryUnit` основан на некоторых предположениях. Во-первых, он не владеет своим прототипом. Можно потребовать, чтобы функция `SetPrototype` удаляла старый прототип перед его переприсваиванием. Во-вторых, класс `PrototypeFactoryUnit` использует функцию `Clone`, предположительно клонирующую объект. В конкретном приложении можно использовать другое имя, руководствуясь собственными вкусами или требованиями, предъявляемыми библиотекой.

Если нужно настроить фабрику, основанную на применении прототипов, достаточно написать шаблонный класс, аналогичный классу `PrototypeFactoryUnit`. Для этого можно применить механизм наследования, сделав класс `PrototypeFactoryUnit` базовым и замещая соответствующие функции. Допустим, что мы решили реализовать функцию `DoCreate` так, чтобы она возвращала нулевой указатель, если указатель на прототип равен нулю.

```
template <class AbstractProduct, class Base>
class MyFactoryUnit
    : public PrototypeFactoryUnit<AbstractProduct, Base>
{
public:
    // Реализуем функцию DoCreate так, чтобы она допускала
    // нулевой указатель на прототип
    AbstractProduct* DoCreate(Type2Type<AbstractProduct>)
    {
        return pPrototype_ ? pPrototype_->Clone() : 0;
    }
};
```

Вернемся к нашей компьютерной игре. Для того чтобы определить конкретную фабрику, нам нужно написать следующий код приложения.

```
// Код приложения
typedef ConcreteFactory
<
    AbstractEnemyFactory,
    PrototypeFactoryUnit
>
EnemyFactory;
```

Итак, дуэт классов AbstractFactory/ConcreteFactory предоставляет нам следующие возможности.

- Легко определять фабрики с помощью списков типов.
- Поскольку класс AbstractFactory наследует каждый из своих компонентов, его интерфейс обладает высокой модульностью. Отдельные указатели или ссылки на подобъекты класса AFUnit<T> можно передавать разным модулям, уменьшая общую взаимозависимость классов.
- Для реализации класса AbstractFactory можно применить класс ConcreteFactory, используя стратегию, диктующую метод создания объектов. Статически связанным стратегиям создания объектов (таким как класс OpNewFactoryUnit, использующий оператор new) нужно передавать список конкретных изделий, создаваемых фабрикой.
- Наиболее распространенной стратегией создания объектов является шаблон проектирования Prototype. Эту стратегию можно легко реализовать в классе ConcreteFactory с помощью шаблонного класса PrototypeFactoryUnit.

Попробуйте получить все перечисленные выше преимущества, реализовав шаблон проектирования Abstract Factory вручную.

9.5. Резюме

Шаблон проектирования Abstract Factory — интерфейс для создания семейства связанных или взаимозависимых полиморфных объектов. Используя этот шаблон, можно разделить классы реализации на разные непересекающиеся семейства.

Интерфейс обобщенной абстрактной фабрики можно реализовать с помощью списков типов и шаблонных стратегий. Списки типов предусматривают списки изделий (конкретных и абстрактных) и шаблонные стратегии.

Шаблонный класс AbstractFactory представляет собой скелет для определения абстрактных фабрик в сочетании с шаблонным классом AFUnit. Класс AbstractFactory использует класс GenScatterHierarchy (глава 3) для генерирования модульного интерфейса, наследующего класс AFUnit<T> для каждого абстрактного изделия T, указанного в списке типов. Эта структура позволяет уменьшить взаимозависимость классов, передавая в различные части приложения только отдельные модули фабрик.

Шаблон ConcreteFactory позволяет реализовать интерфейс класса AbstractFactory. Класс ConcreteFactory использует для создания объектов стратегию FactoryUnit и класс GenLinearHierarchy (глава 3). В библиотеке Loki предусмотрены две реализации стратегии FactoryUnit: класс OnNewFactoryUnit, создающий объекты с помощью оператора new, и класс PrototypeFactoryUnit, создающий объекты с помощью клонирования прототипов.

9.6. Краткий обзор классов AbstractFactory и ConcreteFactory

- Объявление класса AbstractFactory выглядит следующим образом.

```
template
<
   class TList,
```

```
    template <class> class Unit = AbstractFactoryUnit
>
class AbstractFactory;
```

Здесь класс TList является списком типов, состоящим из абстрактных изделий, создаваемых фабрикой, а класс Unit — это шаблон, определяющий интерфейс каждого типа из списка TList. Например, приведенное ниже выражение определяет абстрактную фабрику, способную клонировать объекты классов Soldier, Monster и SuperMonster.

```
typedef AbstractFactory<TIPELIST_3(Soldier, Monster,
    SuperMonster> AbstractEnemyFactory;
```

- Класс AbstractFactoryUnit<T> определяет интерфейс, состоящий из чисто виртуальных функций, имеющих сигнатуру T* DoCreate(Type2Type<Type>). Обычно функцию DoCreate не нужно вызывать явно. Вместо нее вызывается функция AbstractFactory::Create.
- Класс AbstractFactory содержит шаблонную функцию Create, которую можно конкретизировать для любого из типов абстрактных изделий.

```
AbstractEnemyFactory *pFactory = ...;
Soldier *pSoldier = pFactory->Create<Soldier>();
```

- Для реализации интерфейса, определенного классом AbstractFactory, библиотека Loki содержит шаблонный класс ConcreteFactory.

```
template
<
    class AbstractFact,
    template <class, class> class FactoryUnit = OpNewFactoryUnit,
    class TList = AbstractFact::ProductList
>
class ConcreteFactory;
```

Здесь класс AbstractFact представляет собой конкретизацию класса AbstractFactory, подлежащего реализации. Класс FactoryUnit является реализацией стратегии FactoryUnit, а класс TList — это список конкретных изделий.

- Класс FactoryUnit имеет доступ к абстрактному и конкретному изделиям, подлежащим созданию. В библиотеке Loki определены две стратегии Creator: классы OpNewFactoryUnit (раздел 9.3) и PrototypeFactoryUnit (раздел 9.4). Их можно использовать для настраиваемой реализации стратегии FactorUnit.
- Класс OpNewFactoryUnit использует для создания объектов оператор new. Применяя этот класс, нужно передавать список типов, состоящий из типов конкретных изделий, в качестве третьего шаблонного параметра класса ConcreteFactory.

```
typedef ConcreteFactory
<
    AbstractEnemyFactory,
    OpNewFactoryUnit,
    TYPELIST_3(SillySoldier, SillyMonster,
        SillySuperMonster)
>
EasyLevelEnemyFactory;
```

- Класс PrototypeFactoryUnit хранит указатели на типы абстрактных изделий и создает новые объекты, вызывая функцию-член Clone с соответствующими параметрами. Для этого в классе PrototypeFactoryUnit для каждого абстрактного изделия T нужно определить отдельную виртуальную функцию Clone, возвращающую указатель типа T* и дублирующую объект.
- При использовании класса PrototypeFactoryUnit в сочетании с классом ConcreteFactory третий шаблонный аргумент класса ConcreteFactory не нужен.

```
typedef ConcreteFactory
<
   AbstractEnemyFactory,
   PrototypeFactoryUnit
>
EnemyProduct;
```

10

Шаблон Visitor

В этой главе обсуждаются обобщенные компоненты, использующие шаблон проектирования `Visitor` (Gamma et al., 1995). Это мощный шаблон проектирования, изменяющий зависимость между проектными решениями, принятыми при разработке класса.

Шаблон `Visitor` (Инспектор) обеспечивает удивительную гибкость: в иерархию классов можно добавлять виртуальные функции, не прибегая к повторной компиляции ни этих функций, ни существующих клиентских кодов. Однако эта гибкость достигается за счет определенных жертв: в иерархию теперь невозможно добавить новый терминальный класс (leaf class), не повторив компиляцию всей иерархии и ее клиентов. Следовательно, область применения шаблона `Visitor` ограничивается только очень устойчивыми иерархиями (в которые редко добавляются новые классы) и программами, в которые часто приходится добавлять новые виртуальные функции.

Шаблон `Visitor` противоречит программистской интуиции. Следовательно, для его успешного применения нужны тщательная реализация и строгая дисциплина. В главе описывается обобщенная реализация шаблона `Visitor`, оставляющая прикладному программисту очень мало работы.

В главе обсуждаются следующие темы.

- Принцип работы шаблона `Visitor`.
- Когда следует применять шаблон `Visitor` и, что не менее важно, когда его применять не следует.
- Базовая реализация инспектора (предложенная группой GoF).
- Как устранить некоторые недостатки базовой реализации шаблона `Visitor`.
- Как реализовать решения, касающиеся реализации шаблона `Visitor`, в виде библиотеки.
- Мощные обобщенные компоненты, облегчающие реализацию инспекторов.

10.1. Основы шаблона Visitor

Рассмотрим иерархию классов, которую мы хотим наделить новыми функциональными возможностями. Для этого в нее можно добавлять либо новые классы, либо новые виртуальные функции-члены.

Добавить новые классы легко. Можно применить механизм наследования к терминальному классу и реализовать необходимые виртуальные функции. Для этого не нужно изменять или компилировать вновь существующие классы. Их коды остаются неизменными и готовы к повторному использованию.

В противоположность этому, добавить новую виртуальную функцию трудно. Для того чтобы полиморфно манипулировать объектами (с помощью указателей на объекты базового класса), виртуальную функцию-член нужно добавить не только в базовый класс, но и, возможно, во многие другие классы, входящие в иерархию. Это — основная операция. Она модифицирует базовый класс, от которого зависит вся иерархия и ее клиенты. В результате все придется повторно компилировать.

Короче говоря, с точки зрения зависимостей между классами новые классы добавлять легко, а новые виртуальные функции-члены — трудно.

Однако допустим, что у нас есть иерархия, в которую редко добавляются новые классы и в то же время часто добавляются новые виртуальные функции-члены. В этой ситуации возможность легко добавлять новые классы представляет собой ненужное преимущество. Вот тут-то и пригодится шаблон `Visitor`. Этот шаблон предоставляет программисту именно ту функциональную возможность, в которой он в данный момент нуждается, позволяя легко вставлять в иерархию новые виртуальные функции-члены, одновременно затрудняя вставку новых классов. За это придется заплатить всего-навсего лишним виртуальным вызовом.

Шаблон `Visitor` лучше всего проявляет свои возможности, когда операции на объектами не связаны между собой. Парадигма "состояние и операции" для каждого класса становится не совсем уместной. В данной ситуации лучше придерживаться принципа *отделения* типов от операций. Имеет смысл хранить разные реализации одной операции вместе, а не распределять их по иерархии классов.

Допустим, что мы разрабатываем текстовый редактор. Элементы документа, такие как абзацы и графические изображения, представляются в виде классов, производных от одного базового класса, скажем, `DocElement`. Документ — это структурированный набор указателей на объекты класса `DocElement`. Программисту нужна возможность перемещаться по этой структуре и выполнять операции, например, проверку орфографии, переформатирование и вычисление статистических показателей. В идеале следовало бы реализовывать эти операции, добавив новый код без модификации существующей программы. Более того, программу было бы легче сопровождать, если бы весь код, связанный, скажем, с вычислением статистических показателей, хранился в одном месте.

К статистическим показателям могут относиться общее количество символов, значащих символов, слов и рисунков. Все это естественным образом можно поместить в класс `DocStats`.

```
class DocStats
{
  unsigned int,
     chars_,
     nonBlankChars_,
     words_,
     images_;
  ...
public:
 void AddChars(unsigned int charsToAdd)
 {
 chars_ += charsToAdd;
 }
 ... классы AddWords, AddImages и др. определяются так же ...
 // Статистика отображается в диалоговом окне
 void Display();
};
```

Используя классический объектно-ориентированный подход к сбору статистики, можно определить в классе `DocElement` соответствующую виртуальную функцию.

```
class DocElement
{
 ...
 // Эта функция-член предназначена для сбора статистики
 virtual void UpdateStats(DocStats& statistics) = 0;
};
```

В этом случае каждый конкретный элемент документа может определять эту функцию по-своему. Например, классы `Paragraph` и `RasterBitmap`, производные от класса `DocElement`, могут реализовывать функцию `UpdateStats` следующим образом.

```
void Paragraph::UpdateStats(DocStats& statistics)
{
 statistics.AddChars(количество символов в абзаце);
 statistics.AddWords(количество слов в абзаце);
}

void RasterBitmap::UpdateStats(DocStats& statistics)
{
 // Этот класс учитывает только рисунки и больше ничего
 // (символы и другие элементы игнорируются)
 statistics.AddImages(1);
}
```

В итоге функция, управляющая сбором статистических данных, может выглядеть так.

```
void Document::DisplayStatistics()
{
 DocStats statistics;
 for (каждый элемент документа)
 {
 элемент->UpdateStats(statistics);
 }
 statistics.Display();
}
```

Это очень хорошая реализация возможностей по сбору статистики, однако у нее есть ряд недостатков.

- Она требует, чтобы класс `DocElement` и производные от него классы имели доступ к определению функции `DocStats`. Следовательно, при каждой модификации класса `DocStats` придется компилировать заново всю иерархию класса `DocElement`.
- Фактические операции по сбору статистики рассеяны по разным реализациям функции `UpdateStats`. При отладке или расширении возможностей, связанных со сбором статистики, придется искать и редактировать несколько файлов.
- Реализация не масштабируется по мере добавления новых операций, аналогичных сбору статистики. Для того чтобы добавить операцию "увеличить размер шрифта на один пункт", понадобится добавить в класс `DocElement` новую виртуальную функцию, преодолевая все связанные с этим трудности.

Однако связь, существующую между классами `DocElement` и `DocStats`, можно разорвать, если переместить все операции внутрь класса `DocStats` и позволить ему самому определять, какую операцию выполнять для того или иного типа. Для этого нужно, чтобы в

классе DocStats существовала функция-член void UpdateStats(DocElement&). Тогда документ просто просматривал бы свои элементы и вызывал для каждого из них функцию UpdateStats.

Это решение сразу скрывает класс DocStats от класса DocElement. Однако теперь класс DocStats зависит от каждого конкретного объекта класса DocElement, который необходимо обработать. Если иерархия объектов более постоянна, чем множество операций, эта зависимость никому не мешает. Теперь проблема заключается в том, что реализация класса UpdateStats должна использовать так называемое *переключение типов* (type switch). Это переключение возникает при обращении к полиморфному объекту по его конкретному типу и выполнении разных операций в зависимости от этого типа. Функция DocStats::UpdateStats связана с этим переключением типов.

```
void DocStats::UpdateStats(DocElement& elem)
{
 if (Paragraph* p = dynamic_cast<Paragraph*>(&elem))
 {
    chars_ += p->NumChars();
    words_ += p->NumWords();
 }
 else if (dynamic_cast<RasterBitmap*>(&elem))
 {
    ++images_;
 }
 else ...
 добавляем по одному оператору if для каждого типа инспектируемого объекта
}
```

(Определение указателя p внутри оператора if вполне законно, поскольку это мало известное свойство языка C++. Переменную можно определять и проверять непосредственно внутри оператора if. Область видимости этой переменной ограничена оператором if и его частью else. Несмотря на то что эта возможность не представляется нам существенной, и создавать остроумные коды в качестве самоцели не рекомендуется, отметим, что она предназначена именно для переключения типов, так почему бы не воспользоваться этим?)

Когда программист видит что-то подобное, у него в голове должен сразу же сработать предохранитель. Переключение типов не всегда желательно. (В главе 8 приводится детальная аргументация этого утверждения.) Код, основанный на переключении типов, трудно понимать, расширять и сопровождать. Он не защищен от случайных ошибок. Например, что произойдет, если поставить вызов оператора dynamic_cast для базового класса перед вызовом оператора dynamic_cast для производного класса? Во время первой проверки будут сопоставляться производные классы, поэтому вторая проверка никогда не будет выполнена. Одна из целей полиморфизма — избежать переключения типов и связанных с ним проблем.

Посмотрим, в каких ситуациях может оказаться полезным шаблон Visitor. Допустим, нам нужно, чтобы новые функции работали виртуально, но мы не хотим вводить новые виртуальные функции для каждой операции. Для этого нужно реализовать *переключающую виртуальную функцию* (bouncing virtual function), единственную во всей иерархии класса DocElement. Эта функция будет "пересылать" задание в другую иерархию. Иерархия DocElement называется *инспектируемой* (visited), а операции принадлежат новой иерархии *инспектора* (visitor).

Каждая реализация переключающей виртуальной функции вызывает *другую* функцию в иерархии инспектора. Так выбираются инспектируемые типы. Функции в ие-

рархии инспектора, вызываемые переключающей функцией, являются виртуальными. *Так выбираются операции.*

Эту идею иллюстрирует приведенный ниже фрагмент кода. Во-первых, мы определяем абстрактный класс DocElementVisitor, в котором определяется операция для каждого типа объектов в иерархии класса DocElement.

```
class DocElementVisitor
{
public:
 virtual void VisitParagraph(Paragraph&) = 0;
 virtual void VisitRasterBitmap(RasterBirmap&) = 0;
 ... другие подобные функции ...
}
```

Затем в иерархию класса DocElement добавляется переключающая виртуальная функция под названием Accept, получающая параметр DocElementVisitor& и вызывающая соответствующую функцию Visit*Xxx*.

```
class DocElement
{
public:
   virtual void Accept(DocElementVisitor&) = 0;
   ...
};

void Paragraph::Accept(DocElementVisitor& v)
{
   v.VisitParagraph(*this);
}

void RasterBitmap::Accept(DocElementVisitor& v)
{
   v.VisitRasterBitmap(*this);
}
```

А вот и функция DocStats во всей своей красе.

```
class DocStats : public DocElementVisitor
{
public:
 virtual void VisitParagraph(Paragraph& par)
 {
    chars_ += par.NumChars();
    words_ += par.NumWords();
 }

 virtual void VisitRasterBirmap(RasterBirmap&)
 {
    ++images_;
 }
 ...
};
```

(Разумеется, реальную реализацию этой функции следует поместить в самостоятельный файл, отделив ее от определения класса.)

Этот небольшой пример иллюстрирует недостаток шаблона Visitor: виртуальная функция-член на самом деле не добавляется. Настоящие виртуальные функции имеют полный доступ к каждому объекту, для которого они определены, в то время как из функции VisitorParagraph можно получить доступ лишь к открытой части класса Paragraph.

Управляющая функция Document::DisplayStatistics создает объект класса DocStats и вызывает функцию Accept для каждого объекта класса DocElement, передавая его качестве параметра. Поскольку объект класса DocStats инспектирует разные конкретные объекты класса DocElement, он отлично собирает все данные, и переключение типов совсем не нужно.

```
void Document::DisplayStatistics()
{
   DocStats statistics;
   for (каждый элемент документа)
   {
      элемент->Accept(statistics);
   }
   statistics.Display();
}
```

Проанализируем возникающий контекст. Мы создали новую иерархию, корнем которой является класс DocElementVisitor. Это *иерархия операций* (hierarchy of operations). Каждый класс, входящий в нее, на самом деле представляет собой операцию, например DocStats. Теперь добавлять новые операции совсем несложно. Для этого нужно лишь создать новый класс, производный от класса DocElementVisitor. Ни один элемент иерархии класса DocElement для этого менять не требуется.

Например, добавим одну операцию, скажем, IncrementFontSize, связанную с горячей клавишей, предназначенной для увеличения размера шрифта на один пункт, или соответствующей кнопкой.

```
class IncrementFontSize : public DocElementVisitor
{
public:
   virtual void VisitParagraph(Paragraph& par)
   {
      par.SetFontSize(par.GetFontSize() + 1);
   }

   virtual void VisitRasterBitmap(RasterBitmap&)
   {
      // Ничего не делаем
   }
   ...
};
```

Вот в все. Не нужно вносить никаких изменений ни в иерархию класса DocElement, ни в другие операции. Вы просто добавляете новый класс. Функция DocElement::Accept перебирает все объекты класса IncrementFontSize совершенно так же, как объекты класса DocStats. Полученная в результате структура классов представлена на рис. 10.1.

Напомним, что по умолчанию новый класс легко добавить, а виртуальную функцию-член — нет. Мы преобразовали классы в функции с помощью перехода от иерархии класса DocElement к иерархии класса DocElementVisitor. Таким образом, классы, производные от класса DocElementVisitor, представляют собой *воплощенные функции* (objectified functions). Так работает шаблон проектирования Visitor.

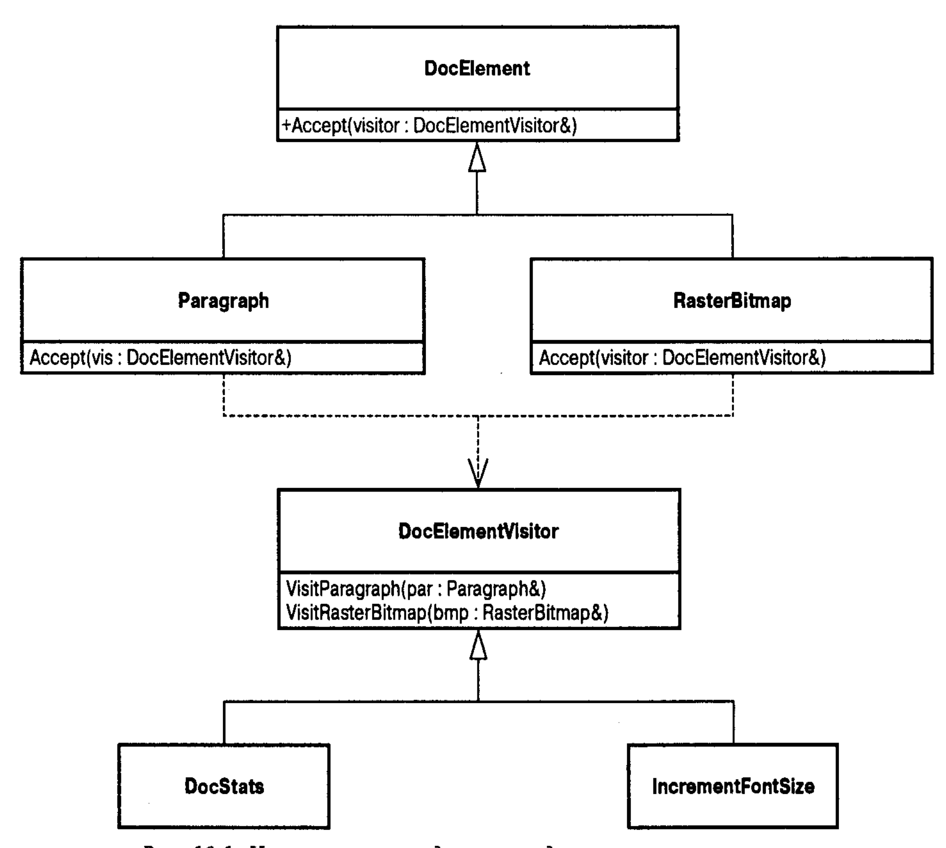


Рис. 10.1. Иерархия классов для игры с двумя уровнями сложности

10.2. Перегрузка и функция-ловушка

Управление перегрузкой функций в языке C++ оказывает очень большое влияние на реализацию проектов на основе шаблона `Visitor`, хотя для самого шаблона перегрузка функций значения не имеет.

В классе `DocElementVisitor` каждому инспектируемому типу соответствует отдельная функция-член: `VisitParagraph(Paragraph&)`, `VisitRasterBitmap(RasterBitmap&)` и т.д. Эти функции явно избыточны. Кроме того, имя инспектируемого типа является частью самой функции.

Как правило, избыточности лучше избегать. Для этого следует применять перегрузку функций. Мы просто назовем все функции одним именем `Visit` и предоставим компилятору решать, какую перегруженную функцию `Visit` вызывать для передаваемого параметра заданного типа. Альтернативное определение класса `DocElementVisitor` выглядит следующим образом.

```
class DocElementVisitor
{
public:
   virtual void Visit(Paragraph&) = 0;
   virtual void Visit(RasterBitmap&) = 0;
   ... другие подобные функции ...
};
```

Теперь можно проще определить функции-члены Accept, поскольку их поведение становится единообразным.

```
void Paragraph::Accept(DocElementVisitor& v)
{
   v.Visit(*this);
}

void RasterBitmap::Accept(DocElementVisitor& v)
{
   v.Visit(*this);
}
```

Они выглядят настолько одинаковыми, что может возникнуть соблазн перенести их в базовый класс DocElement. Это было бы ошибкой. На самом деле это совершенно разные функции. Параметр *this в функции Paragraph::Accept имеет статический тип Paragraph&, а в функции RasterBitmap::Accept — тип RasterBitmap&. Именно эти статические типы помогают компилятору правильно определять, какую из перегруженных функций DocElementVisitor::Visit следует вызывать. Если бы функция Accept была реализована в классе DocElement, параметр *this имел бы статический тип DocElement&, который не может предоставить компилятору всю необходимую информацию. Таким образом, факторизация классов не должна быть произвольной. Неправильная факторизация может привести программу в негодность.

Перегрузка функций порождает интересную идею. Допустим, что все классы, производные от класса DocElement, реализуют функцию Accept, просто переадресовывая вызов функции DocElementVisitor::Visit. Тогда в классе DocElement можно определить следующую перегруженную функцию-ловушку (catch-all overload).

```
class DocElementVisitor
{
public:
  ... как и прежде ...
  virtual void Visit(DocElement&) = 0;
};
```

Когда будет вызываться эта перегруженная функция? Если новый класс является непосредственным наследником класса DocElement и в классе DocElementVisitor для него нет подходящей перегруженной функции Visit, то вступают в силу правила перегрузки и автоматического преобразования производных типов в базовые. Ссылка на объект неизвестного класса автоматически конвертируется в ссылку на объект базового класса DocElement, и вызывается функция-ловушка. Если такой функции нет, возникает ошибка компиляции. Предусматривать функцию-ловушку или нет, зависит от конкретной ситуации.

В перегруженной функции-ловушке можно выполнить много полезной работы. Примеры представлены в работах Влиссидеса (Vlissides, 1998, 1999). В такой функции можно выполнять произвольные действия весьма обобщенного характера или прибегнуть в переключению типов (используя оператор dynamic_cast), чтобы распознать фактический тип параметра DocElement.

10.3. Уточнение реализации: шаблон Acyclic Visitor

Итак, вы решили применить шаблон Visitor. Вас интересует, как это сделать в реальном проекте?

Анализ зависимостей между классами в предыдущем примере приводит нас к следующим выводам.

- Для того чтобы скомпилировать определение класса `DocElement`, необходимо знать о существовании класса `DocElementVisitor`, поскольку он упоминается в сигнатуре функции-члена `DocElement::Accept`. Для этого достаточно сделать неполное объявление класса (forward declaration).
- Для того чтобы скомпилировать определение класса `DocElementVisitor`, необходимо знать о существовании *всех* конкретных классов, входящих в иерархию класса `DocElement`, поскольку имена этих классов встречаются в функциях-членах `Visit`*`Xxx`* класса `DocElementVisitor`.

Такой тип зависимости называется *циклическим* (cyclic dependency). Циклические зависимости создают хорошо известные трудности. Для класса `DocElement` необходим класс `DocElementVisitor`, а классу `DocElementVisitor` нужны все классы из иерархии класса `DocElement`. Следовательно, класс `DocElement` зависит от всех своих подклассов. Фактически здесь проявляется *циклическая зависимость по имени* (cyclic name dependency), т.е. для компиляции определений классов нужны лишь их *имена*. Таким образом, классы следует разделить по файлам.

```
// Файл DocElementVisitor.h
class DocElement;
class Paragraph;
class RasterBitmap;
... неполные объявления всех подклассов класса DocElement ...

class DocElementVisitor
{
   virtual void VisitParagraph(Paragraph&) = 0;
   virtual void VisitRasterBitmap(RasterBitmap&) = 0;
   ... другие подобные функции ...
};

// Файл DocElement.h
class DocElementVisitor;

class DocElement
{
public:
   virtual void Accept(DocElementVisitor&) = 0;
  ...
};
```

Более того, для каждой однострочной реализации функции `Accept` необходимо знать определение класса `DocElementVisitor`, а каждый конкретный инспектор должен быть внедрен в класс, который он должен инспектировать. Все это порождает весьма запутанную схему взаимозависимостей.

Добавление новых подклассов класса `DocElement` становится крайне сложным. Но ведь мы и не собирались добавлять в иерархию класса `DocElement` никаких новых элементов! Шаблон `Visitor` предназначен прежде всего для устойчивых иерархий, в которые добавляются лишь операции, а не новые классы. Стоит напомнить, что шаблон `Visitor` позволяет это сделать за счет усложнения процедуры добавления новых производных классов в инспектируемую иерархию (в нашем примере — в иерархию класса `DocElement`).

Однако жизнь — сложная штука. В реальном мире устойчивых иерархий вообще не бывает. Когда-нибудь вам *понадобится* добавить новый класс в иерархию класса

DocElement. В предыдущем примере для этого придется добавить новый класс VectorGraphic, производный от класса DocElement. Следуйте нашим указаниям.

- Откройте файл DocElementVisitor.h и добавьте новое неполное объявление класса VectorGraphic.
- Включите в класс DocElement новую чисто виртуальную перегруженную функцию.

```
class DocElementVisitor
{
   ... как и прежде ...
   virtual void VisitVectorGraphic(VectorGraphic&) = 0;
};
```

- Реализуйте в каждом конкретном инспекторе функцию VisitVectorGraphic. Эта функция может быть пустой. В качестве альтернативы можно определить пустую функцию DocElementVisitor::VisitVectorGraphic, однако в этом случае компилятор не сообщит ничего, если вы забудете реализовать ее в конкретных инспекторах.
- Реализуйте функцию Accept в классе VectorGraphic. *Сделайте это обязательно.* Если, применяя прямое наследование от класса DocElement, вы забудете реализовать функцию Accept, ничего страшного не произойдет — вы получите сообщение об ошибке компиляции. Однако, если вы выполните наследование от другого класса, например, класса Graphic, в котором определена функция Accept, компилятор не проронит ни слова о том, что вы забыли сделать класс VectorGraphic инспектируемым. Эта ошибка проявится лишь во время выполнения программы, когда вы с удивлением обнаружите, что ни одна реализация функции VisitVectorGraphic не вызывается. Найти такую ошибку достаточно сложно.

Следствие: все иерархии классов DocElement и DocElementVisitor будут скомпилированы повторно, и вам придется добавлять довольно большое количество механического кода лишь для того, чтобы сохранить работоспособность программы. В некоторых случаях такое решение может оказаться совершенно неприемлемым.

Роберт Мартин (Robert Martin, 1996) изобрел интересную разновидность шаблона Visitor, в котором цикличность ликвидируется с помощью оператора dynamic_cast. В рамках этого подхода для иерархии инспектора создается фиктивный базовый класс BaseVisitor, выступающий в роли носителя информации. Он не имеет никакого содержания, а следовательно, совершенно независим. Функция-член Accept из инспектируемой иерархии получает ссылку на объект класса BaseVisitor и применяет к нему оператор dynamic_cast, чтобы обнаружить соответствующий инспектор. Если соответствие достигнуто, функция Accept переходит от инспектируемой иерархии к иерархии инспектора.

Это звучит довольно загадочно, но на самом деле все очень просто. Посмотрим, как можно реализовать шаблон Acyclic Visitor в проекте DocElement/DocElementVisitor. Во-первых, определим базовый класс инспектора.

```
class DocElementVisitor
{
public:
   virtual void ~DocElementVisitor() {}
};
```

Пустой виртуальный деструктор выполняет две важные вещи. Во-первых, он предоставляет в распоряжение класса DocElementVisitor функциональные возможности механизма RTTI (runtime type information). (Оператор dynamic_cast можно применять лишь к тем типам, которые содержат хотя бы одну виртуальную функцию.) Во-

вторых, виртуальный деструктор гарантирует корректное полиморфное уничтожение объектов класса DocElementVisitor. Полиморфная иерархия без виртуальных деструкторов вызывает непредвиденные последствия, если объект производного класса разрушается через указатель на объект базового класса. Таким образом, обе опасности устраняются всего одной строчкой кода.

Определение класса DocElement остается прежним. Для нас представляет интерес определение чисто виртуальной функции Accept(DocElementVisitor&).

Затем для каждого производного класса в инспектируемой иерархии (корнем которой является класс DocElement) определим небольшой инспектирующий класс, имеющий только одну функцию Visit*Xxx*. Например, для класса Paragraph нужно определить следующий класс.

```
class ParagraphVisitor
{
public:
   virtual void VisitParagraph(Paragraph&) = 0;
};
```

Реализация функции Paragraph::Accept выглядит так.

```
void Paragraph::Accept(Visitor& v)
{
   if (ParagraphVisitor* p =
       dynamic_cast<ParagraphVisitor*>(&v))
   {
      p->VisitParagraph(*this);
   }
   else
   {
      здесь можно вызвать функцию-ловушку
   }
}
```

Ведущую роль в этой функции играет оператор dynamic_cast, позволяющий системе поддержки выполнения программ распознать, является ли объект v подобъектом класса ParagraphVisitor, и, если да, получить указатель на этот подобъект.

Аналогичные реализации функции Accept определяются во всех классах, производных от класса DocElement. В заключение конкретный инспектор выводится из класса DocElementVisitor и базовых инспекторов для всех классов, представляющих интерес.

```
class DocStats :
   public DocElementVisitor,   // Необходим
   public ParagraphVisitor,    // Нужен для инспектирования
                               // объектов класса Paragraph
   public RasterBitmapVisitor, // Нужен для инспектирования
                               // объектов класса RasterBitmap
{
public:
   void VisitParagraph(Paragraph& par)
   {
      chars_ += par.NumChars();
      words_ += par.NumWords();
   }
   void VisitRasterBitmap(RasterBitmap&)
   {
      ++images_;
   }
};
```

Структура классов, полученная в результате, показана на рис. 10.2. Горизонтальные пунктирные линии изображают зависимости, существующие в иерархии, а вертикальная — границу между иерархиями. Обратите внимание на то, как оператор dynamic_cast позволяет волшебным образом перескакивать с одного иерархического дерева на другое, используя в качестве пружины фиктивный класс DocElementVisitor.

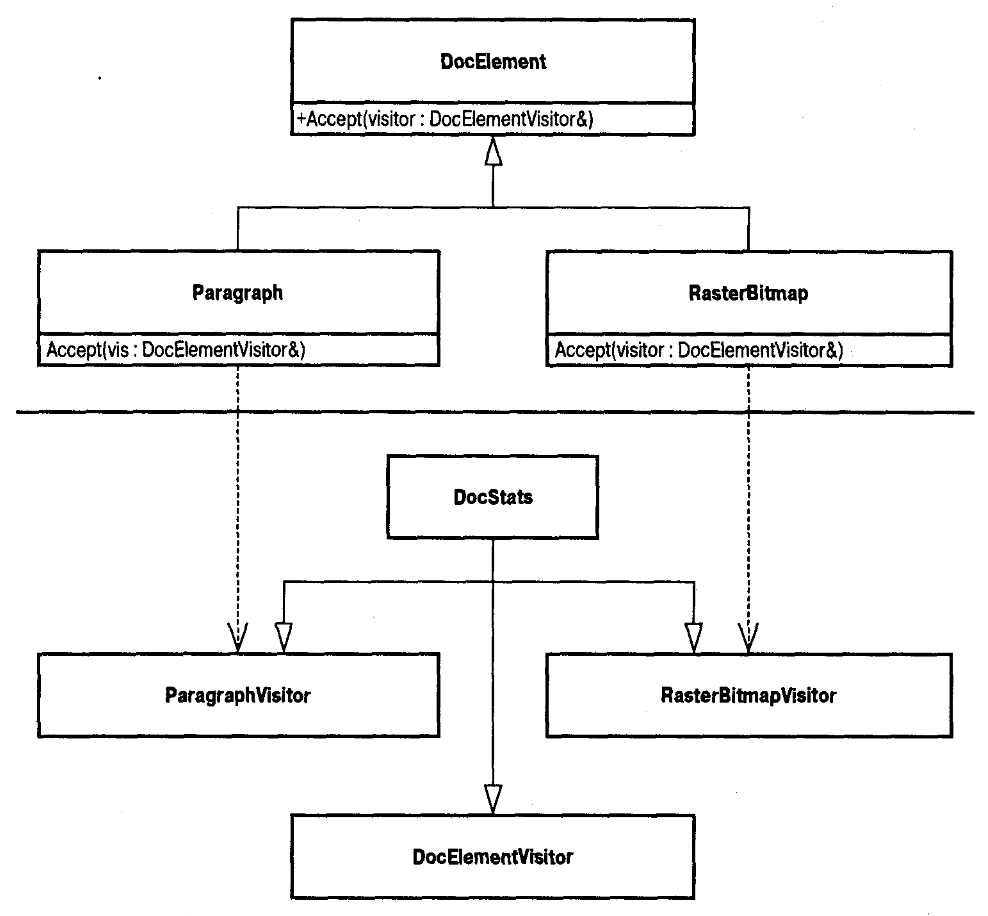


Рис. 10.2. Структура классов для ациклического инспектора

Хотя описанная структура содержит в себе большое количество деталей и взаимодействий, базовая структура довольно проста. Убедимся в этом, проанализировав поток событий. Допустим, что у нас есть указатель pDocElem на объект класса DocElement, имеющего динамический ("реальный") тип Paragraph. Тогда поступим следующим образом.

```
DocStats stats;
pDocElem->Accept(stats);
```

Это повлечет за собой такие события.

1. Объект stats автоматически преобразовывается в ссылку на объект класса DocElementVisitor, поскольку он является открытым наследником этого класса.
2. Вызывается виртуальная функция Paragraph::Accept.

3. Функция Paragraph::Accept пытается применить оператор dynamic_cast<ParagraphVisitor*> к адресу объекта DocElementVisitor, полученного ею в качестве параметра. Поскольку этот объект имеет динамический тип DocStats, который является открытым наследником классов DocElementVisitor и ParagraphVisitor, выполняется приведение типов. (Вот где происходит телепортация!)
4. Теперь функция Paragraph::Accept должна получить указатель на часть класса ParagraphVisitor, которую унаследовал объект DocStats. Функция Paragraph::Accept применяет к этому указателю виртуальную функцию VisitParagraph.
5. Виртуальный вызов достигает функции DocStats::VisitParagraph. Кроме того, она получает в качестве параметра ссылку на инспектируемый абзац. Инспектирование закончено.

Проверим новую диаграмму зависимостей.

- Определение класса DocElement зависит от класса DocElementVisitor по имени. Зависимость по имени означает, что неполного объявления класса DocElementVisitor вполне достаточно.
- Класс ParagraphVisitor — и вообще все базовые классы *Xxx*Visitor — зависят по имени от классов, которые они инспектируют.
- Реализация функции Paragraph::Accept полностью зависит от класса ParagraphVisitor. Полная зависимость означает, что для компиляции кода необходимо полное определение класса.
- Все конкретные определения классов инспекторов полностью зависят от класса DocElementVisitor и всех базовых инспекторов *Xxx*Visitor.

Шаблон Acyclic Visitor ликвидирует циклические зависимости, но взамен оставляет программисту еще больше работы. Теперь мы должны поддерживать *два* параллельных множества классов: инспектируемую иерархию, корнем которой является класс DocElement, и множество инспектирующих классов *Xxx*Visitor, по одному на каждый инспектируемый класс. Работать с двумя параллельными иерархиями классов нежелательно, поскольку для этого требуется строгая дисциплина и внимание.

Сравнивая эффективность “простого” шаблона Visitor и шаблона Acyclic Visitor, следует заметить, что во втором случае при каждом проходе иерархии возникает одно дополнительное динамическое приведение типов. Время, затрачиваемое на это приведение, может быть постоянным, а может зависеть от количества полиморфных классов, возрастая по логарифмическому или линейному закону. Вид этой зависимости зависит от конкретного компилятора. Если эффективность является важным критерием качества программы, то эти затраты времени могут стать существенными. Таким образом, иногда следует использовать “простой” шаблон Visitor и поддерживать циклические связи.

Глядя на эту мрачную картину, следует признать, что шаблон Visitor представляет собой противоречивую конструкцию. Это не удивительно, поскольку даже Ральф Гамма (Ralf Gamma) из группы GoF заявлял, что шаблон Visitor замыкает десятку самых непопулярных шаблонов (Vlissides, 1999).

Шаблон Visitor грубый, негибкий, его трудно расширять и поддерживать. Однако, будучи настойчивыми и прилежными, мы можем создать библиотечную реализа-

цию шаблона Visitor, представляющую собой полную противоположность сказанному выше: ее легко использовать, расширять и поддерживать. Как это сделать, мы покажем в следующем разделе.

10.4. Обобщенная реализация шаблона Visitor

Разделим реализацию на две основные части.

- *Инспектируемые классы.* Это классы, входящие в иерархию, которую мы собираемся инспектировать (добавляя в нее операции).
- *Инспектирующие классы.* Эти классы определяют и реализуют фактические операции.

Наш метод прост: мы стремимся вынести в библиотеку максимальную часть кода. Если нам это удастся, взаимозависимости между классами значительно упростятся. Таким образом, инспектор и инспектируемый будут зависеть не друг от друга, а от библиотеки. Это очень хорошо, поскольку библиотека считается намного более постоянной, чем приложение.

Сначала мы попробуем реализовать обобщенный шаблон Acyclic Visitor, поскольку он обладает лучшими качествами с точки зрения зависимостей между его частями. Позднее мы его усовершенствуем, повысив эффективность. В заключение вернемся к реализации классического шаблона Visitor, предложенного группой GoF, обладающего более высоким быстродействием за счет потери гибкости.

При обсуждении вопросов реализации мы будем пользоваться именами, перечисленными и определенными в табл. 10.1. Некоторые из этих имен на самом деле описывают шаблонные классы, точнее говоря, *становятся* шаблонными классами по мере повышения степени обобщенности нашей программы. Пока нас будут интересовать лишь определения сущностей, к которым относятся эти имена.

Таблица 10.1. Имена компонентов

Имя	Принадлежность	Описание
BaseVisitable	Библиотека	Основа всех инспектируемых иерархий
Visitable	Библиотека	Смешанный класс, благодаря которому класс в инспектируемой иерархии становится инспектируемым
DocElement	Приложение	Основа иерархии, которую мы хотим сделать инспектируемой
Paragraph, RasterBitmap	Приложение	Два конкретных инспектируемых класса, производных от класса DocElement
Accept	Библиотека и приложение	Функция-член, которая вызывается для инспектирования иерархии
BaseVisitor	Библиотека	Основа инспектирующей иерархии
Visitor	Библиотека	Смешанный класс, благодаря которому класс в инспектирующей иерархии становится инспектирующим
Statistics	Приложение	Конкретный инспектирующий класс
Visit	Библиотека и приложение	Функция-член, которая вызывается инспектируемыми элементами и применяется к инспекторам

Сначала обратим внимание на инспектирующую иерархию. Здесь все довольно просто — мы должны предусмотреть некоторые базовые классы для пользователя, т.е. классы, определяющие операцию Visit для заданного типа. Кроме того, необходимо создать фиктивный класс, используемый оператором динамического приведения типов в соответствии с шаблоном Acyclic Visitor.

```
class BaseVisitor
{
public:
   virtual ~BaseVisitor() {}
};
```

Кода для повторного применения здесь немного, но иногда приходится создавать и такие маленькие классы.

Теперь напишем простой шаблонный класс Visitor.

```
template <class T>
class Visitor
{
public:
   virtual void Visit(T&) = 0;
};
```

В общем случае функция Visitor<T>::Visit может возвращать значение, тип которого отличается от типа void. Эта функция может передавать полезный результат через функцию Visitable::Accept. Следовательно, в класс Visitor нужно добавить второй шаблонный параметр.

```
template <class T, typename R = void>
class Visitor
{
public:
   typedef T ReturnType;
   virtual ReturnType Visit(T&) = 0;
};
```

Для того чтобы проинспектировать иерархию, мы должны создать класс ConcreteVisitor, производный от класса BaseVisitor, а количество реализаций класса Visitor должно совпадать с количеством инспектируемых классов.

```
class SomeVisitor :
   public BaseVisitor // Необходим
   public Visitor<RasterBitmap>,
   public Visitor<Paragraph>
{
public:
   void Visit(RasterBitmap&) // Инспектирует класс RasterBitmap
   void Visit(Paragraph&)    // Инспектирует класс Paragraph
};
```

Этот код выглядит простым, ясным и легким в использовании. Его структура четко указывает, какие классы подлежат испектированию. И, что еще лучше, компилятор не позволяет конкретизировать класс SomeVisitor, если мы не определили все функции Visit. В этом проявляется связь между определением класса SomeVisitor и именами инспектируемых им классов (RasterBitmap и Paragraph). Эта зависимость вполне понятна, поскольку класс SomeVisitor *знает* об этих типах и *нуждается* в специальных операциях для работы с ними.

На этом мы завершим обсуждение части реализации, касающейся инспектирующих классов. Мы определили простой базовый класс BaseVisitor, действующий как

оболочка оператора dynamic_cast, и шаблонный класс Visitor, генерирующий чисто виртуальную функцию Visit.

Несмотря на то что код, написанный нами для этого, весьма мал, его потенциал повторного использования огромен. Шаблонный класс Visitor генерирует отдельный тип для каждого инспектируемого класса. Пользователь библиотеки теперь может не разбивать реализацию на такие маленькие классы, как ParagraphVisitor и RasterVisitor. Типы, генерируемые шаблонным классом Visitor, сами обеспечат связь между всеми инспектируемыми классами.

Перейдем к инспектируемой иерархии. Как указывалось в предыдущем разделе, инспектируемая иерархия в реализации шаблона Acyclic Visitor предназначена для решения следующих задач.

- Объявлять чисто виртуальную функцию Accept, получающую ссылку на объект класса Visitor в базовом классе.
- Замещать и создавать реализацию функции Accept в каждом производном классе.

Для того чтобы выполнить первую задачу, мы добавим в класс BaseVisitable чисто виртуальную функцию и потребуем, чтобы класс DocElement наследовал ее. Это относится и ко всем корням инспектируемых иерархий, существующих в приложении. Класс BaseVisitable выглядит следующим образом.

```
template <typename R = void>
class BaseVisitable
{
public:
   typedef R ReturnType;
   virtual ~BaseVisitable() {}
   virtual ReturnType Accept(BaseVisitor&) = 0;
};
```

Все очень просто и скучно. Интересные вещи происходят, лишь когда мы пытаемся выполнить вторую задачу, т.е. реализовать функцию Accept в библиотеке (а не в клиентском коде). Реализация функции Accept невелика, но включать ее в каждый клиентский класс — слишком утомительное занятие. Было бы здорово, если бы этот код принадлежал библиотеке. Как предписывает шаблон Acyclic Visitor, если класс T является инспектируемым, то реализация T::Accept применяет оператор dynamic_cast<Visitor<T>*> к типу BaseVisitor. Если приведение выполнено успешно, функция T::Accept возвращает управление функции Visitor<T>::Visit. Однако, как указывалось в разделе 10.2, простое определение функции Accept в классе BaseVisitable не работает, поскольку параметр *this имеет статический тип BaseVisitable, который не может предоставить инспекторам достаточный объем информации о типе.

Нужно найти способ реализовать функцию Accept в библиотеке, а затем вставить ее в иерархию класса DocElement. Увы, в языке C++ такого непосредственного механизма нет. Есть средства для работы с виртуальным наследованием, однако они работают не лучшим образом. Мы должны прибегнуть к макросу и потребовать, чтобы каждый класс в инспектируемой иерархии использовал этот макрос в своем определении.

Принять решение использовать макрос (со всеми вытекающими отсюда последствиями) нелегко, но другого более удобного и эффективного решения не существует. Поскольку программисты на языке C++ люди практичные, эффективность является достаточной причиной для использования именно макроса, а не другого, эзотерического, но неэффективного средства.

Основное правило гласит: макрос должен выполнять как можно меньше работы и как можно быстрее пересылать данные “реальным” сущностям (функции или классу). Макрос для инспектируемых классов определяется следующим образом.

```
#define DEFINE_VISITABLE() \
   virtual ReturnType Accept(BaseVisitor& guest) \
   { return AcceptImpl(*this, guest); }
```

Пользователь должен вставить макрос DEFINE_VISITABLE() в каждый класс инспектируемой иерархии. Этот макрос переопределяет функцию-член Accept как шаблонную функцию AcceptImpl, параметризованную типом параметра *this. Таким образом, функция AcceptImpl получает доступ к необходимому статическому типу. Функция AcceptImpl определена в самом низу иерархии, в классе BaseVisitor. Это позволяет всем производным классам иметь к ней доступ. Вот как выглядит измененный класс BaseVisitable.

```
template <typename R = void>
class BaseVisitable
{
public:
   typedef R ReturnType;
   virtual ~BaseVisitable() {}
   virtual ReturnType Accept(BaseVisitor&) = 0;
protected: // Открывает доступ к иерархии
   template <class T>
   static ReturnType AcceptImpl(T& visited, BaseVisitor& guest)
   {
   // Применяем шаблон Acyclic Visitor
   if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
   {
      return p->Visit(visited);
   }
   return ReturnType();
 }
};
```

То, что мы отправили функцию AcceptImpl в библиотеку, очень важно. Это сделано не только для автоматизации работы пользователя. Присутствие функции AcceptImpl в библиотеке дает нам возможность настраивать ее реализацию на конкретные условия проекта.

Реализация проекта Visitor/Visitable скрывает от пользователя большое количество деталей, представляя собой волшебный черный ящик. Ниже приведен код реализации обобщенного шаблона Acyclic Visitor.

```
// Инспектирующая часть
class BaseVisitor
{
public:
   virtual ~BaseVisitor() {}
};
template <class T, typename R = void>
class Visitor
{
public:
   typedef R ReturnType; // Доступен для клиентов
   virtual ReturnType Visit(T&) = 0;
};
```

```
// Инспектируемая часть
template <typename R = void>
class BaseVisitable
{
public:
   typedef R ReturnType;
   virtual ~BaseVisitable() {}
   virtual ReturnType Accept(BaseVisitor&) = 0;
protected:
   template <class T>
   static ReturnType AcceptImpl(T& visited, BaseVisitor& guest)
   {
     // Применяем шаблон Acyclic Visitor
     if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
     {
         return p->Visit(visited);
     }
     return ReturnType();
   }
};

#define DEFINE_VISITABLE() \
   virtual ReturnType Accept(BaseVisitor& guest) \
   { return AcceptImpl(*this, guest); }
```

Готовы к тестовым испытаниям? Начнем!

```
class DocElement : public BaseVisitable<>
{
public:
   DEFINE_VISITABLE()
};

class Paragraph : public DocElement
{
public:
   DEFINE_VISITABLE()
};

class MyConcreteVisitor :
   public BaseVisitor,          // Необходим
   public Visitor<DocElement>,  // Инспектирует объекты
                                // класса DocElements
   public Visitor<Paragraph>    // Инспектирует объекты
                                // класса Paragraph
 {
 public:
   void Visit(DocElement&)
     { std::cout << "Visit(DocElement&) \n"; }
   void Visit(Paragraph&)
     { std::cout << "Visit(Paragraph&) \n"; }
 };

int main()
{
   MyConcreteVisitor visitor;
   Paragraph par;
   DocElement* d = &par; // Скрывает статический тип объекта par
   d->Accept(visitor);
}
```

Эта маленькая программа выводит на экран сообщение, означающее "все в порядке".

```
Visit(Paragraph&)
```

Разумеется, этот искусственный пример не может продемонстрировать всю мощь разработанного нами кода. Однако, если вспомнить, с какими трудностями мы столкнулись в предыдущем разделе при реализации инспекторов "с нуля", становится ясно, что теперь в нашем распоряжении есть средство для корректного создания инспектируемых иерархий и их дальнейшего инспектирования.

Перечислим действия, которые необходимо выполнить при определении инспектируемой иерархии.

- Вывести корень иерархии из класса `BaseVisitable<YourReturnType>`.
- Добавить в каждый класс `SomeClass`, входящий в инспектируемую иерархию, макрос `DEFINE_VISITABLE()`. (Теперь иерархию можно инспектировать, однако никаких зависимостей от класса `Visitor` больше нет!)
- Вывести каждый конкретный инспектирующий класс из класса `BaseVisitor`. Кроме того, для каждого класса `X`, подлежащего инспектированию, нужно вывести класс `SomeVisitor` из класса `Visitor<X, YourReturnType>`. Обеспечить замещение функции-члена `Visit` в каждом инспектируемом классе.

Диаграмма зависимостей, возникающая в результате этих действий, очень проста. Определение класса `SomeVisitor` зависит по имени от каждого инспектируемого класса. Реализации функции-члена `Visit` полностью зависят от классов, которыми они манипулируют.

Все это прекрасно. По сравнению с реализациями, обсуждавшимися ранее, лучшего и быть не может. Благодаря реализации шаблона `Visitor` у нас есть упорядоченный способ создания инспектируемых иерархий, позволяющий сократить клиентский код и зависимость классов.

В особых случаях функцию `Accept` лучше реализовывать непосредственно, а не с помощью макроса `DEFINE_VISITABLE()`. Допустим, что мы определяем класс `Section`, производный от класса `DocElement` и содержащий несколько объектов класса `Paragraph`. Мы бы хотели инспектировать все объекты класса `Paragraph`, содержащиеся в объекте класса `Section`. В этом случае мы могли бы реализовать функцию `Accept` вручную.

```
class Section : public DocElement
// Функция Accept реализуется непосредственно,
// а не через макрос DEFINE_VISITABLE()
{
  ...
  virtual ReturnType Accept(BaseVisitor& v)
  {
     for (каждый параграф в данном разделе)
     {
         current_paragraph->Accept(v);
     }
  }
};
```

Очевидно, что с помощью шаблона `Visitor` можно делать *все*, что угодно. Код, приведенный выше, освобождает программиста от тяжелой необходимости создавать все с нуля.

Мы закончили разработку ядра реализации шаблона `Visitor`, содержащего практически все, что необходимо для инспектирования. Продолжение следует.

10.5. Назад — к "простому" шаблону Visitor

Реализация обобщенного шаблона `Acyclic Visitor`, определенная в предыдущем разделе, вполне удовлетворительно работает во многих ситуациях. Однако, если нужно создать быстродействующее приложение, динамическое приведение типов, выполняемое в функции `Accept`, может повергнуть вас в уныние, а измерения скорости работы вашей программы — прямо в состояние депрессии. Почему это происходит? Когда вы применяете оператор `dynamic_cast` к некоторому объекту, система поддержки выполнения программ должна произвести несколько действий. Код механизма RTTI должен определить, допустимо ли данное преобразование, и в случае положительного ответа вычислить указатель на объект результирующего типа.

Попробуем разобраться, как этого можно достичь при создании компилятора. Можно присвоить каждому типу, который фигурирует в программе, уникальный целочисленный идентификатор. Этот идентификатор оказывается полезен и при обработке исключительных ситуаций. Тогда в виртуальную таблицу каждого класса компилятор записывает указатель на таблицу идентификаторов всех его подтипов. Вместе с ними компилятор должен хранить смешения подобъектов относительно базового объекта. Этой информации достаточно для правильного выполнения динамического приведения типов. Когда выполняется оператор `dynamic_cast<T2*>(p1)`, а параметр `p1` представляет собой "указатель на объект типа `T1`", система поддержки выполнения программ просматривает таблицу типов в поисках типа, соответствуюшего типу `T2`. Если соответствие с типом `T2` обнаружено, система поддержки выполнения программ выполнит необходимые арифметические операции с указателем и вернет результат. В противном случае будет возвращен нулевой указатель. Детали, в частности множественное наследование, еще больше усложняют и замедляют динамическое приведение типов.

Сложность описанного выше решения равна $O(n)$, где n — количество базовых классов данного класса. Иными словами, время, затрачиваемое на динамическое приведение типов, возрастает линейно по мере углубления и расширения иерархий наследования.

В качестве альтернативы можно использовать таблицы хэширования, позволяюшие повысить быстродействие программы за счет бо́льших затрат памяти. Кроме того, можно применить большую матрицу для всего приложения. В этом случае время, затрачиваемое на динамическое приведение типов, постоянно, однако размер программы резко возрастает, особенно, если в ней много классов. (Можно было бы сжать матрицу — тогда жизнь создателей компиляторов языка C++ стала бы намного легче.)

Итак, оператор `dynamic_cast` значительно снижает производительность программы, причем предсказать величину этого снижения невозможно. В некоторых случаях это оказывается совершенно неприемлемым. Следовательно, мы должны расширить нашу реализацию шаблона `Visitor`, чтобы адаптировать "циклический" шаблон `Visitor`, предложенный группой GoF. Это позволит повысить быстродействие нашего приложения, так как в "циклическом" шаблоне `Visitor` не используется динамическое приведение типов, хотя поддерживать его труднее.

Работа шаблона `Visitor`, предложенного группой GoF, уже описывалась в начале главы. Ниже перечисляются основные различия между "обычным" шаблоном `Visitor` и шаблоном `Acyclic Visitor`, реализованным в библиотеке Loki.

- Класс `BaseVisitor` больше не является фиктивным. В нем определяется одна чисто виртуальная функция-член `Visit` для каждого инспектируемого типа (в предположении, что мы используем перегрузку функций).

- Функция AcceptImpl должна измениться. В идеале макрос DEFINE_VISITABLE() остается неизменным.

Все это сводится к следующему. У нас есть набор типов, подлежащих инспектированию: например, DocElement, Paragraph и RasterBitmap. Как выразить этот набор типов и манипулировать с ним? Естественно в этот момент на ум приходят списки типов, описанные в главе 3.

Списки типов — это именно то, что нам нужно. Мы хотим передавать список типов шаблонному классу CyclicVisitor в качестве шаблонного параметра, как бы говоря: "Я хочу, чтобы данный класс CyclicVisitor мог инспектировать данные типы". Сделать это можно следующим элегантным способом.

```
// Неполное объявление, необходимое для списка типов
class DocElement;
class Paragraph;
class RasterBitmap;

// Инспектирует классы DocElement, Paragraph и RasterBitmap
typedef CyclicVisitor
<
   void, // Тип возвращаемого значения
   TYPELIST_3(DocElement, Paragraph, RasterBitMap)
>
MyVisitor;
```

Класс CyclicVisitor зависит по имени от классов DocElement, Paragraph и RasterBitmap.

Посмотрим, какие дополнения нужно сделать в нашем коде. Используем процедуру, описанную в главе 9 при определении обобщенной реализации шаблона Abstract Factory.

```
// Определение класса Private::VisitorBinder<R>
// содержится в файле Visitor.h
template <typename R, class TList>
class CyclicVisitor : public GenmScatteredHierarchy<TList,
   Private::VisitorBinder<R>::Result>
{
   typedef R ReturnType;
   template <class Visited>
   ReturnType Visit(Visited& host)
   {
     Visitor<Visited>& subObj = *this;
     return subObj.Visit(host);
   }
};
```

Следует отметить, что класс CyclicVisitor использует класс Visitor в качестве строительного блока. Аналогичный способ был продемонстрирован в главе 3, а похожий пример был рассмотрен в главе 9 По существу, класс CyclicVisitor наследует класс Visitor<T> для каждого типа T, указанного в списке TList.

Если передать классу CyclicVisitor список типов, он завершит наследование от класса Visitor, конкретизированного для каждого типа, указанного в данном списке, объявив таким образом по одной чисто виртуальной функции Visit для каждого типа. Иными словами, с функциональной точки зрения он эквивалентен базовому классу Visitor, как это предписывает шаблон Visitor, предложенный группой GoF.

После уточнения класса CyclicVisitor с помощью оператора typedef, скажем, классом MyVisitor, нам остается лишь применить макрос DEFINE_CYCLIC_VISITABLE(MyVisitor) в соответствующих инспектируемых классах.

```
typedef CyclicVisitor
<
   void, // Тип возвращаемого значения
   TYPELIST_3(DocElement, Paragraph, RasterBitMap)
>
MyVisitor;

class DocElement
{
public:
   virtual void Visit(MyVisitor&) = 0;
};

class Paragraph : public DocElement
{
public:
   DEFINE_CYCLIC_VISITABLE(MyVisitor);
};
```

Вот и все! Как и в классической реализации шаблона Visitor, предложенной группой GoF, нужно лишь быть дисциплинированным. Отличие заключается в том, что теперь вам придется держать в голове намного меньше информации. Для того чтобы сделать иерархию инспектируемой в рамках реализации "простого" шаблона Visitor, нужно выполнить следующие действия.

- Сделать неполное объявление всех классов, входящих в иерархию.
- Написать оператор typedef для класса CyclicVisitor, конкретизированного типом возвращаемого значения и списком инспектируемых типов. (Назовем этот класс MyVisitor.)
- Определить в базовом классе чисто виртуальную функцию Visit.
- Добавить макрос DEFINE_CYCLIC_VISITABLE(MyVisitor) в каждый класс иерархии или реализовать функцию Accept вручную.
- Сделать каждый конкретный инспектор наследником класса MyVisitor.
- Обновлять конкретизацию шаблонного класса MyVisitor (оператор typedef) каждый раз, когда в иерархию инспектируемых классов добавляется новый класс, и — увы! — компилировать ее снова.

По сравнению с реализацией "обычного" шаблона Visitor обобщенный подход более понятен. Программисту нужно лишь самостоятельно определить класс MyVisitor.

С практической точки зрения лучше начинать с шаблона Acyclic Visitor, который легче поддерживать, а на шаблон Visitor следует переходить лишь тогда, когда оптимизация становится действительно необходимой. Обобщенные компоненты позволяют легко экспериментировать — для этого нужно изменить лишь одно объявление. Остальной код остается без изменения. Детали реализации шаблона Visitor хранятся в библиотеке. Нужно лишь наладить связь между клиентом и библиотекой, чтобы получить доступ к двум разным реализациям шаблона Visitor.

10.6. Отладка вариантов

Шаблон Visitor имеет много вариантов и настроек. В этом разделе мы покажем, как его можно применять в собственных приложениях.

10.6.1. Функция-ловушка

Эта функция рассматривалась в разделе 10.2. Класс Visitor может столкнуться с неизвестным типом, производным от базового класса (например, классом DocElement). В этом случае либо компилятор выдаст сообщение об ошибке, либо во время выполнения программы будут произведены какие-то действия, предусмотренные по умолчанию.

Посмотрим, как решается эта проблема в реализациях шаблонов Visitor и AcyclicVisitor с помощью обобщенных компонентов.

Для "обычного" шаблона Visitor ситуация проста. Если базовый класс вашей иерархии входит в список типов, передаваемых классу CyclicVisitor, существует возможность реализовать функцию-ловушку. В противном случае возникнет ошибка компиляции. Эти две возможности иллюстрируются следующим кодом.

```
// Неполные объявления, необходимые шаблону "обычному" Visitor
class DocElement; // Базовый класс
class Paragraph;
class RasterBitmap;
class VectorizedDrawing;

typedef CyclicVisitor
<
   void, // Тип возвращаемого значения
   TYPELIST_3(Paragraph, RasterBitmap, VectorizedDrawing)
>
StrictVisitor; // Операции перехвата не предусмотрены.
               // При попытке инспектирования неизвестного
               // типа возникает ошибка компиляции
typedef CyclicVisitor
<
   void, // Тип возвращаемого значения
   TYPELIST_4(DocElement,Paragraph, RasterBitmap,
      VectorizedDrawing)
>
NonStrictVisitor; // Объявляет функцию Visit(DocElement&),
                  // которая будет вызываться при попытке
                  // проинспектировать неизвестный тип
```

Все это довольно просто. Теперь рассмотрим функцию-ловушку в обобщенной реализации шаблона Acyclic Visitor.

Здесь применяется интересный прием. Хотя по существу перехват касается инспектирования неизвестного класса известным инспектором, проблема переворачивается: неизвестный инспектор инспектирует известный класс!

Вернемся к реализации функции AcceptImpl для шаблона Acyclic Visitor.

```
template <typename R = void>
class BaseVisitable
{
   ... как и раньше ...
   template <class T>
   static ReturnType AcceptImpl(T& visited, BaseVisitor& guest)
```

```
   {
      if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
      {
         return p->Visit(visited);
      }
      return ReturnType(); // Внимание!
   }
};
```

Допустим, мы добавляем в иерархию класса `DocElement` класс `VectorizeDrawing`, ничего не сообщая об этом конкретным инспекторам. При попытке проинспектировать класс `VectorizeDrawing` динамическое приведение к типу `Visitor<VectorizeDrawing>` завершится аварийно, поскольку классу `Visitor` ничего не известно о классе `VectorizeDrawing`. Следовательно, код активирует альтернативную процедуру и вернет значение типа `ReturnType`, предусмотренное по умолчанию. Именно в этом месте вступает в действие функция-ловушка.

Поскольку наша функция `AcceptImpl` содержит операцию `return ReturnType()`, простора для вариаций не остается. Здесь следует применить стратегию перехвата.

```
template
<
   typename R = void.
   template <typename, class> class CatchAll = DefaultCatchAll
>
class BaseVisitable
{
   ... как и раньше ...
  template <class T>
   static ReturnType AcceptImpl(T& visited, BaseVisitor& guest)
   {
      if (Visitor<T>* p = dynamic_cast<Visitor<T>*>(&guest))
      {
         return p->Visit(visited);
      }
      // Изменение
      return CatchAll<R, T>::OnUnknownVisitor(visited, guest);
   }
};
```

Стратегия `CanchAll` вполне соответствует требованиям шаблона. Она может возвращать значение по умолчанию или код ошибки, генерировать исключительную ситуацию, вызывать виртуальную функцию-член для инспектируемого объекта, пытаться выполнять динамическое приведение типов и т.д. Реализация функции `OnUnknownVisitor` значительно зависит от нужд конкретного приложения. В некоторых случаях необходимо, чтобы инспектирование было выполнено для всех типов, входящих в иерархию. В других случаях это относится лишь к некоторым типам, а остальные можно проигнорировать. В реализации шаблона `Acyclic Visitor` в основном применяется второй подход, поэтому по умолчанию стратегия `CatchAll` имеет следующий вид.

```
template <class R, class Visited>
struct DefaultCatchAll
{
   static R OnUnknownVisitor(Visited&, BaseVisitor&)
   {
      // Здесь указываются действия, которые следует
      // предпринять при несоответствии между инспектором и
      // и инспектируемым объектом. Обычно должно
```

```
        // возвращаться значение, заданное по умолчанию
        return R();
    }
};
```

Если нужно, чтобы выполнялись другие действия, достаточно заменить шаблон в классе `BaseVisitable`.

10.6.2. Нестрогое инспектирование

Любой программист хотел бы иметь быстрые, независимые и гибкие инспекторы, способные читать его мысли и отличать обычную ошибку от вынужденного решения. С другой стороны, программисты — люди практичные, и с ними можно договориться.

Гибкость стратегии `CatchAll` вызывает зависть у пользователей шаблона `Visitor`, предложенного группой GoF. Реализация этого шаблона весьма строга — она объявляет одну чисто виртуальную функцию для каждого инспектируемого типа. Каждую перегрузку функции `Visit` нужно выводить из класса `BaseVisitor` и реализовывать отдельно. Если это не сделать, код не будет компилироваться.

Однако иногда инспектировать все типы, указанные в списке, совершенно не обязательно. Программисты хотят иметь выбор: либо реализовывать функцию `Visit` для каждого типа, либо автоматически вызывать функцию `OnUnknownVisitor` для собственной реализации стратегии `CatchAll`.

Для таких ситуаций библиотека Loki предусматривает класс `BaseVisitorImpl`. Он является производным от класса `BaseVisitor` и использует адаптированные списки типов. Его реализацию можно найти в библиотеке Loki (файл `Visitor.h`).

10.7. Резюме

В главе обсуждался шаблон `Visitor` и связанные с ним проблемы. По существу, шаблон `Visitor` позволяет добавлять виртуальные функции в иерархию классов без модификации этих классов. В некоторых случаях шаблон `Visitor` позволяет делать программы более ясными и гибкими.

Однако с шаблоном `Visitor` связано много проблем. Дело в том, что его можно применять лишь для очень устойчивых иерархий классов, для остальных иерархий это сделать практически невозможно. В этом случае можно воспользоваться шаблоном `Acyclic Visitor`, пожертвовав эффективностью приложения.

Используя тщательно продуманные и сложные приемы, можно создать обобщенную реализацию шаблона `Visitor`. Сохраняя практически все преимущества шаблона `Visitor`, его обобщенная реализация позволяет избежать многих проблем.

В конкретных приложениях может оказаться полезным шаблон `Acyclic Visitor`, правда, за счет снижения быстродействия программы. Если скорость работы приложения очень важна, следует использовать обобщенную реализацию шаблона `Visitor`, облегчающую поддержку приложения (нужно контролировать только один класс) и не замедляющую компиляцию.

Обобщенная реализация использует самые совершенные приемы программирования на языке C++, такие как динамическое приведение типов, списки типов и частичная специализация. В результате появляется возможность выделить наиболее общие и повторяющиеся части реализации шаблона `Visitor` в отдельную библиотеку.

10.8. Краткий обзор обобщенных компонентов шаблона Visitor

- Для реализации шаблона Acyclic Visitor используются классы BaseVisitor (в качестве фиктивного базового класса), Visitor и Visitable.

```
class BaseVisitor;

template <class T, typename R = void>
class Visitor;

template
<
   typename R = void,
   template<class, class> CatchAll = DefaultCatchAll
>
class BaseVisitable;
```

- Первый шаблонный параметр классов Visitor и BaseVisitor задает тип значений, возвращаемых функциями-членами Visit и Accept, соответственно. По умолчанию это — тип void (как в большинстве примеров, приведенных в книге GoF (2000), и описаний шаблона Visitor).
- Вторым шаблонным параметром класса BaseVisitable является стратегия перехвата (раздел 10.2).
- Основу инспектируемой иерархии следует выводить из класса BaseVisitable. Затем следует применять макрос DEFINE_VISITABLE() в каждом классе иерархии или предусмотреть собственную реализацию функции Accept(BaseVisitors&).
- Конкретные инспектирующие классы следует выводить из класса BaseVisitor. Кроме того, их можно выводить из класса Visitor<T> для каждого инспектируемого типа T.
- Для "обычного" шаблона Visitor следует применять шаблонный класс CyclicVisitor:

```
template <typename R, class TList>
class CyclicVisitor;
```

- Инспектируемые типы нужно указывать в шаблонном аргументе TList.
- Класс CyclicVisitor можно использовать наравне с классическим классом Visitor.
- Если нужно реализовать лишь отдельную часть шаблона Visitor (нестрогий вариант), инспектируемую иерархию следует выводить из класса BaseVisitorImpl. Этот класс реализует все перегрузки функции Visit, необходимые для вызова функции OnUnknownVisitor. Часть этого класса можно замещать собственными функциями.

11

Мультиметоды

В этой главе определяются, обсуждаются и реализуются мультиметоды в контексте языка C++.

Механизм виртуальных функций в языке C++ позволяет производить диспетчеризацию вызова в зависимости от динамического типа объекта. Мультиметоды предназначены для диспетчеризации вызова функции в зависимости от типов *нескольких* объектов. Для универсальной реализации языка необходима специальная поддержка. Именно так осуществляется реализация языков CLOS, ML, Haskell и Dylan. Однако в языке C++ такой поддержки нет, поэтому его эмуляция остается в компетенции разработчиков библиотек.

В главе обсуждаются типичные решения и некоторые обобщенные реализации каждого из них. Эти решения представляют собой разные компромиссы между быстродействием, гибкостью и управлением зависимостями. Чтобы описать способ диспетчеризации вызова функции в зависимости от нескольких объектов, мы используем термин *мультиметод* (multimethod), позаимствованный у языка CLOS, и *множественная диспетчеризация* (multiple dispatch). В частности, множественная диспетчеризация для двух объектов называется *двойной диспетчеризацией* (double dispatch).

Реализация мультиметодов — это чрезвычайно сложная и увлекательная задача, лишившая сна и покоя многих разработчиков и программистов.[1]

В главе рассматриваются следующие вопросы.

- Определение мультиметодов.
- Идентификация ситуаций, в которых необходимо применять полиморфизм мультиобъектов.
- Обсуждение и реализация трех двойных диспетчеров, представляющих воплощения разных компромиссов.
- Усовершенствование механизма двойной диспетчеризации.

Прочитав эту главу, вы легко справитесь с типичными ситуациями, вынуждающими применять мультиметоды. Кроме того, вы сможете использовать и расширять некоторые надежные обобщенные компоненты, предоставленные библиотекой Loki, которые реализуют мультиметоды.

Мы ограничимся обсуждением мультиметодов для двух объектов (двойная диспетчеризация). Овладев этой методикой, вы сможете самостоятельно распространить ее

---

[1] Если вам не удастся реализовать мультиметоды, рассматривайте эту главу в качестве снотворного средства, хотя я надеюсь, что она не обладает усыпляющим эффектом.

на любое количество объектов. Однако в большинстве ситуаций можно обойтись двойной диспетчеризацией, непосредственно воспользовавшись библиотекой Loki.

11.1. Что такое мультиметоды?

В языке C++ полиморфизм, по существу, означает, что вызов данной функции связан с разными реализациями и зависит от статического или динамического контекста.

В языке C++ реализованы два типа полиморфизма.

- Статический полиморфизм (compile-time polymorphism), осуществляемый с помощью перегрузки и шаблонных функций.[2]
- Динамический полиморфизм (run-time polymorphism)м, реализуемый виртуальными функциями.

Перегрузка — это простой вид полиморфизма, позволяющий нескольким одноименным функциям существовать в одной и той же области видимости. Если эти функции имеют разные списки параметров, компилятор может различать их во время компиляции. Перегрузка представляет собой удобную синтаксическую конструкцию, позволяющую сократить текст программы.

Шаблонные функции являются основой статического механизма диспетчеризации. Они осуществляют более сложный статический полиморфизм.

Имя виртуальной функции связывается с ее конкретной реализацией во время выполнения программы в зависимости от динамического типа объекта, к которому она применяется.

Посмотрим, как масштабируются эти три вида полиморфизма для нескольких объектов. Для перегруженных и шаблонных функций это происходит вполне естественно. Оба механизма допускают использование нескольких параметров, а выбор функции производится на основе запутанных статических правил.

К сожалению, виртуальные функции — единственный механизм, реализующий динамический полиморфизм в языке C++, — могут работать только с одним объектом. Даже синтаксис вызова виртуальной функции — `obj.Fun(`*аргументы*`)` — отдает предпочтение объекту `obj` перед аргументами. (Фактически объект `obj` является лишь одним из аргументов функции `Fun`, доступ к которому осуществляется с помощью указателя `*this`. В языке Dylan, например, оператор "точка" является лишь одним из многих конкретных воплощений общего механизма вызова функции.)

Мы будем называть *мультиметодами*, или *множественной диспетчеризацией*, механизм, связывающий вызов с разными конкретными функциями в зависимости от динамических типов нескольких объектов, участвующих в вызове. Поскольку статический мультиобъектный полиморфизм уже существует, остается только реализовать его динамический аналог.

11.2. Когда нужны мультиметоды

Определить, когда следует использовать мультиметоды, очень просто. Допустим, в программе есть операция, манипулирующая несколькими полиморфными объектами

---

[2] Автоматическое преобразование типов можно было бы квалифицировать как простейший вид полиморфизма, поскольку оно, например, позволяет вызывать функцию `sin::sin` с параметром, имеющим тип `int`, хотя изначально параметр этой функции имеет тип `double`. Однако это мнимый полиморфизм, поскольку к параметрам обоих типов применяется одна и та же функция.

с помощью указателей или ссылок на их базовые классы. Необходимо модифицировать операцию в соответствии с динамическими типами этих объектов.

Столкновения (collisions) представляют собой типичную категорию проблем, которые лучше всего решать с помощью мультиметодов. Предположим, что мы разрабатываем видеоигру, в которой движущиеся объекты выводятся из абстрактного класса `GameObject`. Нам хотелось бы, чтобы их столкновение зависело от типов сталкивающихся объектов: космического корабля с астероидом, корабля с космической станцией или астероида со станцией.[3]

Допустим, что мы хотим выделять области перекрытия двух нарисованных объектов. Напишем программу для рисования, позволяющую пользователю определять прямоугольники, круги, эллипсы, многоугольники и другие фигуры. В основу разработки положим классический объектно-ориентированный подход: определим абстрактный класс `Shape` и выведем все конкретные фигуры из него. Затем будем управлять рисованием с помощью набора указателей на объекты, производных от класса `Shape`.

Тут к нам подходит клиент и просит предусмотреть следующее: если две фигуры пересекутся, область их пересечения нужно нарисовать иначе, чем сами фигуры, например, заштриховать (рис. 11.1).

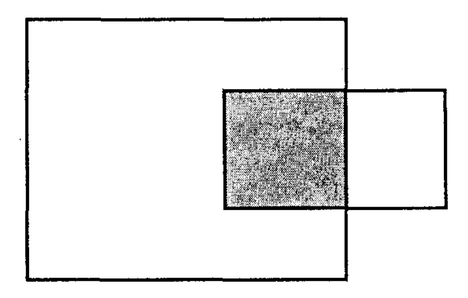


Рис. 11.1. Штриховка пересечения двух фигур

Разработать один алгоритм, заштриховывающий любое пересечение, сложно. Например, алгоритм для штриховки пересечения эллипса и прямоугольника намного сложнее, чем алгоритм для штриховки пересечения двух прямоугольников. Кроме того, чрезмерно общий алгоритм (например, на уровне работы с пикселями) будет слишком неэффективен, поскольку некоторые пересечения (скажем, двух прямоугольников) тривиальны.

Здесь нужен набор алгоритмов, специализированных для каждой пары фигур, например, прямоугольник-прямоугольник, прямоугольник-многоугольник, многоугольник-многоугольник, эллипс-многоугольник и эллипс-эллипс. В ходе выполнения программы, когда пользователь перемещает фигуры по экрану, нужно выбрать правильный алгоритм, который бы закрашивал пересечение фигур достаточно быстро.

Поскольку манипуляции фигурами осуществляются с помощью указателей на класс `Shape`, у нас нет информации о типах объектов, позволяющей выбирать нужный алгоритм. Придется работать только с указателями на класс `Shape`. Поскольку в каждом пересечении участвуют два объекта, простые виртуальные функции не решат нашу задачу. Следует применить двойную диспетчеризацию.

---

[3] Этот пример и имена позаимствованы из книги Скотта Мейерса “More Effective C++” (1996a), задача 31.

11.3. Двойное переключение по типу: грубый подход

Проще всего реализовать двойную диспетчеризацию на основе двойного переключения по типу (double switch-on-type). Для этого нужно попытаться последовательно выполнить динамическое приведение типа первого объекта к каждому из типов объектов, которые могут стоять в левой части выражения. Для каждой ветви нужно сделать то же самое со вторым аргументом. После обнаружения типов для обоих объектов становится ясно, какую функцию следует вызвать.

```
// Разные алгоритмы закраски пересечения фигур
void DoHatchArea1(Rectangle&, Rectangle&);
void DoHatchArea2(Rectangle&, Ellipse&);
void DoHatchArea3(Rectangle&, Poly&);
void DoHatchArea4(Ellipse&, Poly&);
void DoHatchArea5(Ellipse&, Ellipse&);
void DoHatchArea6(Poly&, Poly&);

void DoubleDispatch(Shape& lhs, Shape& rhs)
{
   if (Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea1(*p1, *p2);
      else if (Ellipse p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea2(*p1, *p2);
      else if (Poly p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea3(*p1, *p2);
      else
         Error("Неопределенное пересечение");
   }
   else if (Ellipse* p1 = dynamic_cast<Ellipse*>(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea2(*p2, *p1);
      else if (Ellipse p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea5(*p1, *p2);
      else if (Poly p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea4(*p1, *p2);
      else
         Error("Неопределенное пересечение");
   }
   else if (Poly* p1 = dynamic_cast<Poly*)(&lhs))
   {
      if (Rectangle* p2 = dynamic_cast<Rectangle*>(&rhs))
         DoHatchArea2(*p2, *p1);
      else if (Ellipse p2 = dynamic_cast<Ellipse*>(&rhs))
         DoHatchArea4(*p2, *p1);
      else if (Poly p2 = dynamic_cast<Poly*>(&rhs))
         DoHatchArea6(*p1, *p2);
      else
         Error("Неопределенное пересечение");
   }
   else
   {
         Error("Неопределенное пересечение");
   }
}
```

Надо же! Оказалось, достаточно нескольких строк. Очевидно, что подобный грубый подход вынуждает нас писать большое количество внешне тривиальных кодов. Такую паутину из условных операторов может сплести любой программист. Кроме того, это решение оказывается достаточно эффективным, если количество классов не слишком велико. Функция DoubleDispatch выполняет линейный поиск во множестве возможных типов. Поскольку поиск разворачивается (программа содержит последовательность операторов is-else, а не цикл), для небольших множеств алгоритм будет работать достаточно быстро.

Грубый подход порождает неприемлемый размер программы. При большом количестве классов программу становится невозможно поддерживать. Приведенный выше код работает лишь с тремя классами, но уже имеет значительный размер. С увеличением количества классов размер кода возрастает по экспоненциальному закону. Представьте себе эту программу, если иерархия состоит из 20 классов!

Вторая проблема заключается в том, что реализация функции DoubleDispatch должна иметь информацию о существовании всех классов, входящих в иерархию.

Третий недостаток функции DoubleDispatch заключается в том, что порядок следования операторов if имеет большое значение. Это очень сложная и опасная проблема. Вообразите, например, что из класса Rectangle выводится класс RoundedRectangle (прямоугольник с закругленными углами). Мы редактируем функцию DoubleDispatch, вставляя дополнительные операторы if в конец каждого оператора is-else, непосредственно перед вызовом функции Error. Вот и ошибка!

Если функции DoubleDispatch передается указатель на класс RoundedRectangle, оператор dynamic_cast<Rectangle*> выполняется успешно. Поскольку эта проверка выполняется до проверки dynamic_cast<RoundedRectangle*>, первая проверка "проглотит" и класс Rectangle, и класс RoundedRectangle. Вторая проверка становится излишней. Большинство компиляторов это совершенно не волнует.

Эти проверки следует немного изменить.

```
void DoubleDispatch(Shape& lhs, Shape& rhs)
{
   if (typeid(lhs) == typeid(Rectangle))
   {
      Rectangle* p1 = dynamic_cast<Rectangle*>(&lhs);
      ...
   }
   else ...
}
```

Теперь проверка выполняется только для точного типа, а не для точного или производного. Сравнение typeid выдаст отрицательный результат, если указатель lhs ссылается на объект класса RoundedRectangle, поэтому проверки будут продолжены. В итоге проверка typeid(RoundedRectangle) будет выполнена успешно.

Увы, исправляя один недостаток, мы создаем другой: функция DoubleDispatch становится слишком жесткой. Если в ней не предусмотрен какой-нибудь тип, естественно ожидать, что функция DoubleDispatch сработает на ближайшем к нему базовом типе. Именно на это мы обычно рассчитываем — по умолчанию производные объекты представляют собой базовые объекты, за исключением некоторых замещенных свойств. Проблема заключается в том, что реализация функции DoubleDispatch на основе оператора typeid этой возможностью не обладает. Отсюда следует, что в функции DoubleDispatch нужно по-прежнему использовать оператор dynamic_cast и упорядочить проверки if, чтобы наиболее далекие потомки базового класса обрабатывались первыми.

Так возникают еще два недостатка грубого подхода к реализациям мультиметодов. Во-первых, зависимость функции DoubleDispatch от иерархии класса Shape увеличивается — функция должна знать не только о существовании классов, но и об их взаимоотношениях. Во-вторых, поддержка соответствующей систематизации динамических приведений типов связана с дополнительной нагрузкой.

11.4. Автоматизированный грубый подход

Поскольку в некоторых ситуациях быстродействие грубого подхода оказывается непревзойденным, имеет смысл реализовать такой диспетчер.

Напомним, что в библиотеке Loki определены списки типов — наборы структур и статических алгоритмов, позволяющих манипулировать коллекциями типов. Грубая реализация мультиметодов может использовать предоставленный пользователем список типов, в котором перечислены классы, входящие в иерархию (в нашем примере — классы Rectangle, Poly, Ellipse и др.). Затем рекурсивный шаблон может генерировать последовательность операторов if-else.

В общем случае мы можем работать с разными коллекциями типов, поэтому список типов для левого операнда может отличаться от списка типов для правого операнда.

Попробуем набросать шаблонный класс StaticDispatcher, выполняющий алгоритм вывода типов и вызывающий функцию из другого класса.

```
template
<
   class Executor,
   class BaseLhs,
   class TypeLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs,
   typename ResultType = void
>
class StaticDispatcher
{
   typedef typename TypesLhs::Head Head;
   typedef typename TypesLhs::Tail Tail;
public:
   static ResultType Go(BaseLhs&, BaseRhs&,
      Executor exec)
   {
      if (Head* p1 = dynamic_cast<Head*>(&lhs))
      {
         return StaticDispatcher<Executor, BaseLhs,
            NullType, BaseRhs, TypesRhs>::DispatchRhs(
               *p1, rhs, exec);
      }
      else
      {
         return StaticDispatcher<Executor, BaseLhs,
            Tail, BaseRhs, TypesRhs>::Go(
               lhs, rhs, exec);
      }
   }
   ...
};
```

Класс StaticDispatcher устроен довольно просто. Учитывая сложность задачи, которую он решает, размер его кода удивительно мал.

Класс StaticDispatcher имеет шесть шаблонных параметров. Класс Executor — эти тип объекта, выполняющего реальную работу, — в нашем примере он закрашивает область пересечения фигур. Его внутреннее устройство мы рассмотрим немного позднее.

Классы BaseLhs и BaseRhs представляют собой базовые типы аргументов левого и правого операндов соответственно. Классы TypeLhs и TypeRhs — это списки типов, состоящие из возможных производных классов для этих двух аргументов. По умолчанию классы BaseRhs и TypesRhs реализуют диспетчер для типов, входящих в одну и ту же иерархию, как в программе для рисования.

Класс ResultType — это тип результата двойной диспетчеризации. В общем случае вызываемая функция может возвращать произвольный тип. Класс StaticDispatcher поддерживает такую степень обобщенности и возвращает результат вызывающему модулю.

Функция StaticDispatcher::Go выполняет динамическое приведение адреса lhs к первому типу, указанному в списке TypesLhs. Если динамическое приведение оказалось невыполнимым, функция Go передает хвост списка TypesLhs своему рекурсивному вызову. (На самом деле это не настоящий рекурсивный вызов, поскольку каждый раз производится новая конкретизация класса StaticDispatcher.)

В итоге функция Go выполняет подходящий оператор if-else, применяющий оператор dynamic_cast к каждому типу, указанному в списке. Обнаружив соответствие, функция Go вызывает функцию DispatchRhs. Это второй и последний этап вывода типа — обнаружение динамического типа объекта rhs.

```
template <...>
class StaticDispatcher
{
   ..., как и раньше ...
   template <class SomeLhs>
   static ResultType DispatchRhs(SomeLhs& lhs, BaseRhs& rhs,
      Executor exec)
   {
      typedef typename TypesRhs::Head Head;
      typedef typename TypesRhs::Tail Tail;

      if (Head* p2 = dynamic_cast<Head*>(&rhs))
      {
         return exec.Fire(lhs, *p2);
      }
      else
      {
         return StaticDispatcher<Executor, SomeLhs,
            NullType, BaseRhs, Tail>::DispatchRhs(
               lhs, rhs, exec);
      }
   }
};
```

Функция DispatchRhs выполняет для объекта rhs тот же алгоритм, который применялся функцией Go для объекта lhs. Кроме того, если динамическое приведение типа rhs выполнено успешно, функция DispatchRhs вызывает функцию Executor::Fire, передавая ему два определенных типа. Как и прежде, код, генерируемый компилятором, представляет собой набор операторов is-else. Интересно, что компилятор генерирует для *каждого* типа, указанного в списке TypesLhs, отдельный набор операторов is-else. Действительно, класс StaticDispatcher порождает экспоненциальный объем кода, имея два

списка типов и фиксированный базовый код. Это ценное, однако потенциально опасное качество — слишком большой код может потребовать слишком много времени на этапе компиляции, негативно повлиять на размер программы и общее время ее выполнения.

Для того чтобы установить ограничения на статическую рекурсию, нам нужно специализировать класс `StaticDispatcher` для двух ситуаций, когда тип объекта `lhs` в списке `TypesLhs` и тип объекта `rhs` в списке `TypesRhs` не найдены.

Первая ситуация (неверный объект `lhs`) возникает при вызове функции `Go` из класса `StaticDispatcher`, когда в качестве параметра `TypesList` ему передается класс `NullType`. Это признак того, что во время поиска список `TypesLhs` оказался исчерпанным. (Напомним, что класс `NullType` используется в качестве признака конца списка типов.)

```
template
<
   class Executor,
   class BaseLhs,
   class BaseRhs,
   class TypesRhs,
   typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, NullType,
   BaseRhs, TypesRhs, ResultType>
{
   static void Go(BaseLhs& lhs, BaseRhs& rhs, Executor exec)
   {
      exec.OnError(lhs, rhs);
   }
};
```

Обработка ошибок элегантно делегируется классу `Executor`, который будет рассмотрен ниже.

Вторая ситуация (неверный объект `rhs`) возникает при вызове функции `DispatchRhs` из класса `StaticDispatcher`, когда в качестве параметра `TypesRhs` ему передается класс `NullType`.

```
template
<
   class Executor,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs,
   class TypesRhs,
   typename ResultType
>
class StaticDispatcher<Executor, BaseLhs, TypesLhs,
   BaseRhs, NullType, ResultType>
{
public:
   static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
      Executor& exec)
   {
      exec.OnError(lhs, rhs);
   }
};
```

Теперь настало время обсудить, как нужно реализовать класс `Executor`, чтобы использовать преимущество механизма двойной диспетчеризации, определенного выше.

Класс `StaticDispatcher` лишь распознает типы. Обнаружив правильные типы и объекты, он передает их функции `Executor::Fire`. Чтобы различать вызовы этой

функции, класс Executor должен реализовать несколько перегрузок функции Fire. Например, класс Executor для закрашивания области пересечения двух фигур выглядит следующим образом.

```
class HatchingExecutor
{
public:
   // Разные алгоритмы закрашивания области пересечения
void Fire(Rectangle&, Rectangle&);
void Fire(Rectangle&, Ellipse&);
void Fire(Rectangle&, Poly&);
void Fire(Ellipse&, Poly&);
void Fire(Ellipse&, Ellipse&);
void Fire(Poly&, Poly&);

// Функция для обработки ошибок
void OnError(Shape&, Shape&);
};
```

Класс HatchingExecutor используется вместе с классом StaticDispatcher.

```
typedef StaticDispatcher<HatchingExecutor, Shape,
   TYPELIST_3(Rectangle, Ellipse, Poly)> Dispatcher;
Shape *p1 = ...;
Shape *p2 = ...;
HatchingExecutor exec;
Dispatcher::Go(*p1, *p2, exec);
```

Этот код вызывает соответствующую перегрузку функции Fire из класса HatchingExecutor. Шаблонный класс StaticDispatcher можно рассматривать как механизм динамической перегрузки — он откладывает перегрузку на период выполнения программы. Это значительно облегчает использование класса StaticDispatcher. Нужно лишь реализовать класс HatchingExecutor, имея в виду правила перегрузки, а затем использовать класс StaticDispatcher в качестве черного ящика, применяющего правила перегрузки во время выполнения программы.

Класс StaticDispatcher имеет побочный эффект — он распознает и перегружает неоднозначности на этапе компиляции. Допустим, что мы вместо функции HatchingExecutor::Fire(Ellipse&, Poly&) объявили функцию HatchingExecutor::Fire(Shape&, Poly&). Вызов функции HatchingExecutor::Fire с параметрами типа Ellipse и Poly может породить неоднозначность — обе функции соответствуют одному и тому же вызову. Интересно, что класс StaticDispatcher также обнаруживает эту ошибку и выдает о ней такое же детальное сообщение. Класс StaticDispatcher полностью соответствует правилам перегрузки, существующим в языке C++.

Что случится, если во время выполнения программы обнаружится ошибка — например, если в качестве одного из аргументов функции StaticDispatcher::Go передать класс Circle? Как уже упоминалось, в таких ситуациях класс StaticDispatcher просто вызывает функцию Executor::OnError, параметрами которой являются исходные (не преобразованные) объекты lhs и rhs. Это значит, что в нашем примере функция HatchingExecutor::OnError(Shape&, Shape&) представляет собой модуль обработки ошибок. Эту функцию можно применять по своему усмотрению — ее вызов означает, что класс StaticDispatcher приступает к поиску динамических типов.

Как указывалось в предыдущем разделе, при грубой реализации диспетчера наследование порождает новые проблемы. Следовательно, приведенная ниже конкретизация класса StaticDispatcher содержит ошибку.

```
typedef StaticDispatcher
<
   SomeExecutor,
   Shape,
   TYPELIST_4(Rectangle, Ellipse, Poly, RoundedRectangle)
>
MyDispatcher;
```

Если шаблонному классу MyDispatcher передается класс RoundedRectangle, он будет рассматриваться как класс Rectangle. Оператор dynamic_cast<Rectangle*> успешно применяется к указателю на класс RoundedRectangle, а поскольку оператор dynamic_cast<RoundedRectangle*> находится в цепочке проверок ниже, он никогда не будет выполнен. Правильная конкретизация выглядит следующим образом.

```
typedef StaticDispatcher
<
   SomeExecutor,
   Shape,
   TYPELIST_4(RoundedRectangle, Ellipse, Poly, Rectangle)
>
Dispatcher;
```

Общее правило таково: наиболее отдаленные потомки базового типа должны находиться в начале списка типов.

Было бы прекрасно, если бы это преобразование выполнялось автоматически. Списки типов позволяют это сделать. У нас есть средство для обнаружения наследования во время компиляции (глава 2), причем списки типов можно упорядочить. Таким образом, можно воспользоваться статическим алгоритмом DerivedToFront, описанным в главе 3.

Для автоматической сортировки списка нужно лишь модифицировать реализацию класса StaticDispatcher следующим образом.

```
template <...>
class StaticDispatcher
{
   typedef typename DerivedToFronf<
      typename TypeLhs::Head>::Result Head;
   typedef typename DerivedToFront<
      typename TypesLhs::Tail>::Result Tail;
public:
   ... как и раньше ...
};
```

Обеспечив удобную автоматизацию, не забудьте, что мы получаем в результате программу, генерирующую код. Проблема, связанная с существованием зависимостей между классами, остается нерешенной. Несмотря на то что грубую реализацию мультиметодов очень легко выполнить, класс StaticDispatcher по-прежнему зависит от всех типов, входящих в иерархию. Его преимущества — это быстродействие (если количество классов в иерархии не слишком велико) и ненавязчивость (nonintrusiveness), которая выражается в том, что для использования класса StaticDispatcher не нужно модифицировать иерархию типов.

11.5. Симметричность грубого подхода

Штриховка пересечения двух фигур может зависеть от того, как именно пересекаются фигуры: прямоугольник накрывает эллипс или наоборот. Иногда это не имеет

значения, и штриховка в обоих случаях должна оставаться одинаковой. В этом случае нужен *симметричный мультиметод* (symmetric multimethod), который не зависит от порядка следования его аргументов.

Симметрия применяется, только если типы обоих параметров идентичны (в нашем случае класс BaseLhs совпадает с классом BaseRhs, а LhsTypes совпадает с классом RhsTypes).

Класс StaticDispatcher, реализованный в рамках грубого подхода, является асимметричным. Это значит, что он не поддерживает симметричные мультиметоды. Допустим, например, что мы определяем следующие классы.

```
class HatchingExecutor
{
public:
   void Fire(Rectangle&, Rectangle&)
   void Fire(Rectangle&, Ellipse&)
   ...
   // Обработчик ошибок
   void OnError(Shape&, Shape&);
};

typedef StaticDispatcher
<
   HatchingExecutor,
   Shape,
   TYPELIST_3(Rectangle, Ellipse, Poly)
>
HatchingDispatcher;
```

Класс HatchingDispatcher не срабатывает, если ему передать в качестве левого параметра класс Ellipse, а в качестве правого параметра — класс Rectangle. Даже несмотря на то, что с точки зрения класса HatchingExecutor не имеет никакого значения, какой из этих параметров является левым, а какой — правым, класс HatchingDispatcher настаивает, чтобы объекты передавались в определенном порядке.

Поменяв аргументы местами и применив другую перегрузку в клиентском коде, можно исправить этот недостаток.

```
class HatchingExecutor
{
public:
   void Fire(Rectangle&, Ellipse&);
   // Обеспечиваем симметрию
   void Fire(Ellipse& lhs, Rectangle& rhs)
   {
      // Переходим к функции Fire(Rectangle&, Ellipse&),
      // меняя порядок аргументов
      Fire(rhs, lhs);
   }
   ...
};
```

Эта небольшая функция обеспечивает симметричность мультиметода.

В идеале класс StaticDispatcher должен был бы сам предоставлять эту возможность с помощью дополнительного булевского шаблонного параметра. Для этого нужно, чтобы в некоторых ситуациях класс StaticDispatcher менял порядок следования аргументов при обратном вызове. В каких именно ситуациях? Проанализируем предыдущий пример. Дополняя список шаблонных аргументов их значениями, заданными по умолчанию, мы получаем следующую конкретизацию.

```
typedef StaticDispatcher
<
   HatchingExecutor,
   Shape,
   TYPELIST_2(Rectangle, Ellipse, Poly), // Класс TypesLhs
   Shape,
   TYPELIST_2(Rectangle, Ellipse, Poly), // Класс TypesRhs
   void
>
HatchingDispacther;
```

В симметричном диспетчере можно применить следующий алгоритм выбора пар параметров: объединим первый тип из первого списка типов (класс TypeLhs) с каждым типом из второго списка (класс TypesRhs). Это порождает три комбинации: Rectangle-Rectangle, Rectangle-Ellipse и Rectangle-Poly. Затем объединим второй тип из списка TypesLhs с типами из списка TypesRhs. Однако, поскольку первая комбинация (Rectangle-Ellipse) уже была создана на первом этапе, на этот раз алгоритм начинается со второго элемента в списке TypesRhs. Теперь порождаются пары Ellipse-Ellipse и Ellipse-Poly. Те же рассуждения применяются на следующем шаге: объединение класса Poly из списка TypesLhs с типами из списка TypesRhs начинается с третьего элемента. На этом этапе порождается только одна комбинация — Poly-Poly.

Следуя этому алгоритму, мы реализуем функции только для полученных комбинаций.

```
class HatchingExecutor
{
public:
   void Fire(Rectangle&, Rectangle&);
   void Fire(Rectangle&, Ellipse&);
   void Fire(Rectangle&, Poly&);
   void Fire(Ellipse&, Ellipse&);
   void Fire(Ellipse&, Poly&);
   vois Fire(Poly&, Poly&);
   // Обработчик ошибок
   void OnError(Shape&, Shape&);
};
```

Класс StaticDispatcher должен распознавать комбинации, исключенные нами из описанного выше алгоритма, а именно: Ellipse-Rectangle, Poly-Rectangle и Poly-Ellipse. Для этих комбинаций класс StaticDispatcher должен менять аргументы местами, а в остальных случаях поступать, как и прежде.

Какое булевское условие позволяет определять, когда следует менять аргументы местами, а когда нет? Описанный выше алгоритм выбирает в списке TL2 лишь типы, имеющие индексы, ***равные или превышающие*** индекс типа в списке TL1. Следовательно, условие формулируется следующим образом.

> Если индекс типа U в списке TypesRhs меньше, чем индекс типа T в списке TypesLhs, аргументы следует поменять местами.

Допустим, что типом T является класс Ellipse, а типом U — класс Rectangle. Тогда индекс типа T в списке TypesLhs равен 1, а индекс типа U в списке TypesThs равен 0. Следовательно, аргументы типа Ellipse и Rectangle перед вызовом функции Executor::Fire следует поменять местами.

В набор функциональных возможностей списков типов уже входит статический алгоритм IndexOf, возвращающий позицию типа в списке. Таким образом, сформулировать условие перестановки параметров довольно легко.

Во-первых, мы должны добавить новый шаблонный параметр, который позволяет определить, является ли диспетчер симметричным. Затем нужно добавить небольшой шаблонный класс характеристик InvocationTraits, который либо переставляет аргументы, либо оставляет их на месте перед вызовом функции-члена Executor:Fire.

```
template
<
   class Executor,
   bool symmetric,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs,
   typename ResultType = void
>
class StaticDispatcher
{
   template <bool swapArgs, class SomeLhs, class SomeRhs>
   struct InvocationTraits
   {
      static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
         Executor& exec)
      {
         exec.Fire(lhs, rhs);
      }
   };
   template <class SomeLhs, class SomeRhs>
   struct InvocationTraits<True, SomeLhs, SomeRhs>
   {
      static void DoDispatch(SomeLhs& lhs, SomeRhs& rhs,
         Executor& exec)
      {
         exec.Fire(rhs, lhs); // Меняем аргументы местами
      }
   };
public:
   static void DispatchRhs(BaseLhs& lhs, BaseRhs& rhs,
      Executor exec)
   {
      if (Head* p2 = dynamic_cast<Head*>(&rhs))
      {
         enum { swapArgs = symmetric &&
            IndexOf<Head, TypesRhs>::result <
            IndexOf<BaseLhs, TypesLhs>::result };
         typedef InvocationTraits<swapArgs, BaseLhs, Head>
            CallTraits;
         return CallTraits::DoDispatch(lhs, *p2);
      }
      else
      {
         return StaticDispatcher<Executor, BaseLhs,
            NullType, BaseRhs, Tail>::DispatchRhs(
               lhs, rhs, exec);
      }
   }
};
```

Поддержка симметрии немного усложняет класс StaticDispatch, но намного облегчает его использование.

11.6. Логарифмический двойной диспетчер

Для того чтобы избежать глубоких зависимостей между классами, возникающих при грубой реализации двойного диспетчера, нужно поискать более динамичный подход. Вместо генерирования кода во время компиляции необходимо применить динамические структуры и алгоритмы, обеспечивающие диспетчеризацию вызовов функции во время выполнения программы в зависимости от типов ее параметров.

Решить эту задачу позволяет механизм RTTI (runtime type information), который дает возможность не только идентифицировать типы и выполнять их динамическое приведение, но и систематизировать их в ходе выполнения программы с помощью функции-члена `before` из класса `std::type_info`. Эта функция упорядочивает отношения между всеми типами, определенными в программе, позволяя осуществлять их быстрый поиск.

Реализация такого класса является усовершенствованным решением задачи 31 из книги Скотта Мейерса "More effective C++" (1996а): этап предварительных вычислений (casting step) автоматизирован, а реализация максимально обобщена.

Воспользуемся классом `OrderedTypeInfo`, описанным в главе 2. Этот класс представляет собой оболочку, обладающую функциональными возможностями класса `std::type_info`. Кроме того, он имеет семантику значений и оператор "меньше". Это позволяет хранить объекты класса `OrderedTypeInfo` в стандартных контейнерах. Именно эта особенность представляет для нас интерес.

Подход, предложенный Мейерсом, прост: для каждой пары объектов класса `std::type_info`, подлежащих диспетчеризации, регистрируется указатель на функцию с двойным диспетчером. Этот диспетчер хранит информацию в объекте класса `std::map`. Во время выполнения программы, когда двойному диспетчеру в качестве параметров передаются два неизвестных объекта, он выполняет быстрый поиск типа (затрачивая логарифмическое время) и в случае успеха возвращает соответствующий указатель на функцию.

Определим структуру обобщенного механизма, основанного на этих принципах. Этот механизм представляет собой шаблонный класс с двумя аргументами (левым и правым). Назовем его `BasicDispatcher`, поскольку мы будем использовать его в качестве основы для создания более сложных двойных диспетчеров.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
    typename ResultType = void>
class BasicDispatcher
{
    typedef std::pair<OrderedTypeInfo, OrderedTypeInfo>
        KeyType;
    typedef ResultType (*CallbackType)(BaseLhs&, BaseRhs&);
    typedef CallbackType MappedType;
    typedef std::map<KeyType, MappedType> MapType;
    MapType callbackMap_;
public:
    ...
};
```

В качестве ключей карты используются объекты класса `std::pair`, состоящие из двух объектов класса `OrderedTypeInfo`. Класс `std::pair` поддерживает систематизацию, поэтому для него нам не нужен свой собственный функтор.

Класс `BasicDispatcher` станет еще более общим, если сделать тип обратного вызова шаблонным. Обычно обратный вызов не обязан быть функцией. Он может быть, например, функтором (глава 5). Класс `BasicDispatcher` может адаптировать функторы, преобразовывая определение их внутреннего типа `CallbackType` в шаблонный параметр.

Значительное усовершенствование заключается в следующем: тип `std::map` изменен и стал более эффективным. Мэтт Остерн (Matt Austern, 2000) выяснил, что стандартные ассоциативные контейнеры имеют более узкую область применения, чем принято думать. В частности, упорядоченный вектор в сочетании с алгоритмами бинарного поиска (например, алгоритмом `std::lower_bound`) может оказаться намного эффективнее, чем ассоциативный контейнер. Это происходит, когда количество обращений к его элементам намного превышает количество вставок. Итак, следует внимательно изучить ситуации, в которых обычно применяются объекты двойного диспетчера.

Чаще всего двойной диспетчер представляет собой структуру, в которую информация редко записывается, но из которой часто считывается. Обычно программа лишь однажды устанавливает обратные вызовы, а затем очень часто использует диспетчер. Это хорошо согласуется с использованием механизма виртуальных функций, который дополняется двойным диспетчером. Решение о том, какие функции являются виртуальными, а какие нет, принимается на этапе компиляции.

На первый взгляд упорядоченный вектор нам совершенно не подходит. Операции вставки и удаления элементов такого вектора выполняются за линейное время, и двойной диспетчер никак не может повлиять на быстродействие этих операций. Зато этот вектор позволяет повысить скорость просмотра элемента примерно в два раза и намного снизить объем рабочего множества. Эти свойства очень полезны для двойной диспетчеризации.

Библиотека Loki предусматривает использование упорядоченных векторов, определяя шаблонный класс `AssocVector`, который может заменять собой класс `std::map` (он содержит те же самые функции-члены), реализованный на основе класса `std::vector`. Класс `AssocVector` отличается от класса `std::map` функциями `erase` (функции `AssocVector::erase` аннулируют все итераторы внутри объекта). Кроме того, функции-члены `insert` и `erase` усложнены (выполняя операции вставки и удаления за линейное, а не постоянное время). Поскольку класс `AccosVector` совместим с классом `std::map`, для описания структуры, содержащейся в двойном диспетчере, мы будем также использовать термин *ассоциативный массив* (map).

Ниже приводится новое определение класса `BasicDispatcher`.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicDispatcher
{
   typedef std::pair<TypeInfo, TypeInfo>
      KeyType;
   typedef CallbackType MappedType;
   typedef AssocVector<KeyType, MappedType> MapType;
   MapType callbackMap_;
public:
   ...
};
```

Легко определить и регистрирующую функцию.

```
template <...>
class BasicDispatcher
{
   ... как и раньше ...
```

```
    template <class SomeLhs, SomeRhs>
    void Add(CallbackType fun)
    {
       const KeyType key(typeid(SomeLhs), typeid(SomeRhs));
       callbackMap_[key] = fun;
    }
};
```

Классы SomeLhs и SomeRhs представляют собой конкретные типы, для которых выполняется диспетчеризация вызова. Как и класс std::map, класс AssocVector перегружает оператор [] для доступа к ключу, который соответствует типу, хранящемуся в ассоциативном массиве. Оператор [] возвращает ссылку на новый или найденный элемент, а функция Add связывает с ним параметр fun.

Рассмотрим пример, в котором используется функция Add.

```
typedef BasicDispatcher<Shape> Dispatcher;
// Заштриховывает пересечение прямоугольника и многоугольника
void HatchingRectanglePoly(Shape& lhs, Shape& rhs)
{
   Rectangle& rc = dynamic_cast<Rectangle&>(lhs);
   Poly& pl = dynamic_cast<Poly&>(rhs);
   ... используем ссылки rc и pl ...
}
...
Dispatcher disp;
disp.Add<Rectangle, Poly>(HatchRectanglePoly);
```

Функция-член, выполняющая поиск типа и вызов, довольно проста.

```
template <...>
class BasicDispathcer
{
   ... как и раньше ...
   ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
   {
      MapType::iterator i = callbackMap_.find(
         KeyType(typeid(lhs), typeid(rhs));
      if (i == callbackMap_.end())
      {
         throw std::runtime_error("Функция не найдена");
      }
      return (i->second)(lhs, rhs);
   }
};
```

11.6.1. Логарифмический диспетчер и наследование

Класс BasicDispatcher неправильно работает с механизмом наследования. Если в нем зарегистрирована только функция HatchRectanglePoly(Shape& lhs, Shape& rhs), то диспетчеризация распространяется лишь на объекты классов Rectangle и Poly. Если, например, передать функции BasicDispatcher::Go ссылки на объекты классов RoundedRectangle и Poly, то класс BasicDispatcher откажется выполнять вызов.

Поведение класса BasicDispatcher не согласуется с правилами наследования, в соответствии с которыми производные типы по умолчанию должны рассматриваться как базовые. Было бы прекрасно, если бы класс BasicDispatcher принимал вызовы, параметрами которых являются объекты производных классов, поскольку эти вызовы вполне однозначны.

Для решения этой проблемы нужно не так уж много, однако до сих пор она не ликвидирована окончательно. Таким образом, при регистрации всех пар типов в классе `BasicDispatcher` следует проявлять осторожность.[4]

11.6.2. Логарифмический диспетчер и приведение типов

Класс `BasicDispatcher` полезен, однако не вполне хорош. Зарегистрированная в нем функция, обрабатывающая пересечение между объектами классов `Rectangle` и `Poly`, должна принимать аргументы базового типа `Shape&`. Предлагать клиентскому коду (реализации класса `HatchRectanglePoly`) привести ссылки на объекты класса `Shape` к правильному типу не совсем удобно и безопасно.

С другой стороны, ассоциативный массив обратных вызовов не может хранить разные функции и типы функторов для каждого элемента, поэтому следует стремиться к их единообразному представлению. В параграфе 31 книги Мейерса "More Effective C++" (Meyers, 1996a) эта проблема уже обсуждалась. Преобразование "указатель на функцию — указатель на функцию" не подходит, поскольку после возвращения из функции `FnDoubleDispatcher::Add` информация о статическом типе теряется, и к какому типу следует приводить параметр, неизвестно. (Попробуйте разобрать какой-нибудь код, и вы сразу поймете, в чем дело.)

Мы решим задачу динамического приведения типов с помощью простого обратного вызова функций (а не функторов). Таким образом, шаблонный аргумент `CallbackTemplate` является указателем на функцию.

Решить задачу нам поможет *трамплинная функция* (trampoline function), также известная под названием *санк* (thunk). Трамплинные функции — это маленькие функции, выполняющие небольшую настройку перед вызовами других функций. Обычно они используются разработчиками компиляторов языка C++ для реализации ковариантных типов возвращаемых значений и настройки указателей при множественном наследовании.

Мы будем использовать трамплинную функцию, выполняющую соответствующее приведение типов, а затем вызывать функцию с подходящей сигнатурой, облегчая таким образом работу пользователей. Однако существует одна проблема: объект `callbackMap_` теперь должен хранить *два* указателя на функции: один предоставляется пользователем, а другой является указателем на трамплинную функцию. Это снижает быстродействие программы. Вместо одного косвенного вызова через указатель у нас есть два. Кроме того, реализация усложняется.

Для преодоления этого препятствия воспользуемся следующим фактом. Шаблон может получать указатель на функцию в качестве параметра, лишенного типа. (В большинстве книг шаблонные параметры, лишенные типа, представляют собой целочисленные значения.) Вообще говоря, в качестве параметров, лишенных типа, шаблону можно передавать указатели на любые глобальные объекты, в том числе и функции. При этом должно выполняться единственное условие: функция, адрес которой является шаблонным аргументом, должна иметь внешнюю связь (external linkage). Статические функции можно легко превратить в функции с внешней связью, удалив ключевое слово `static` и поместив их в безымянное пространство имен. Например, в стандартном пространстве имен языка C++ можно объявить следующую функцию.

```
static void Fun();
```

---

[4] Я убежден, что решение этой проблемы существует. Однако, увы, срок работы над книгой ограничен контрактом.

В безымянном пространстве имен эта функция объявляется иначе.

```
namespace
{
   void Fun();
}
```

Используя указатель на функцию в качестве шаблонного параметра, лишенного типа, мы отказываемся хранить его в ассоциативном массиве. Этот важный момент следует хорошо уяснить. Мы поступаем так потому, что компилятор обладает статической информацией об этом указателе. Следовательно, компилятор может зафиксировать адрес функции в трамплинном коде.

Эту идею можно реализовать в новом классе, использующем класс BasicDispathcer в качестве внутреннего буфера (back end). Новый класс FnDispatcher предназначен для диспетчеризации только функций, а не функторов. Он включает класс BasicDispatcher в свой закрытый раздел и предоставляет соответствующие интерфейсные функции.

Шаблонная функция FnDispatcher::Add имеет три шаблонных параметра. Два из них представляют собой левый и правый типы, для которых регистрируется диспетчеризация (классы ConcreteLhs и ConcreteRhs). Третий шаблонный параметр (callback) является указателем на функцию. Дополнительная функция FnDispatcher::Add перегружает определенную выше шаблонную функцию Add, имеющую только два параметра.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   ResultType = void>
class FnDispatcher
{
   BaseDispatcher<BaseLhs, BaseRhs, ResultType> backEnd_;
   ...
public:
   template<class ConcreteLhs, class ConcreteRhs,
      ResultType (*callback)(ConcreteLhs&, ConcreteRhs&)>
   void Add()
   {
     struct Local // см. главу 2
     {
       static ResultType Trapmoline(BaseLhs& lhs, BaseRhs& rhs)
       {
          return callback(
             dynamic_cast<ConcreteLhs&>(lhs),
             dynamic_cast<ConcreteRhs&>(rhs));
       }
   };
   return backEnd_.Add<ConcreteLhs, ConcreteRhs>(
      &Local::Trampoline);
};
```

Используя локальную структуру, мы определяем трамплинную функцию непосредственно внутри функции Add. Трамплинная функция осуществляет приведение аргументов к соответствующим типам, а затем передает управление функции, на которую ссылается указатель обратного вызова. Функция Add добавляет трамплинную функцию в объект callbackMap_ с помощью функции Add объекта backEnd_ (определенной в классе BaseDispatcher).

С точки зрения быстродействия трамплинная функция не создает дополнительных проблем. Хотя она выглядит как косвенный вызов, это не так. Как указывалось выше, компилятор фиксирует адрес, хранящийся в указателе обратного вызова, в коде функции Trampoline, поэтому второй косвенный вызов не возникает. Более изощренные

компиляторы при малейшей возможности делают трамплинную функцию подставляемой.

Использовать новую функцию Add легко.

```
typedef FnDispatcher<Shape> Dispatcher;

// Возможно в безымянном пространстве имен
void HatchRectanglePoly(Rectangle& lhs, Poly& rhs)
{
   ...
}

Dispatcher disp;
disp.Add<Rectangle, Poly, Hatch>();
```

Благодаря функции-члену Add класс FnDispatcher легко использовать. Он также позволяет при необходимости обращаться к функции Add, определенной в классе BaseDispatcher.[5]

11.7. Класс FnDispatcher и симметрия

Благодаря динамизму класса FnDispatcher добавить в него поддержку симметрии намного проще, чем в статический класс StaticDispatcher.

Для этого достаточно зарегистрировать две трамплинные функции: одну — для вызова исполняющей функции при обычном порядке следования аргументов, а вторую — для перестановки параметров перед вызовом. Добавим в функцию Add новый шаблонный параметр.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   typename ResultType = void>
class FnDispatcher
{
   ...
   template <class ConcreteLhs, class ConcreteRhs,
      ResultType (*callback)(ConcreteLhs&, ConcreteRhs&),
      bool symmetric>
   bool Add()
   {
      struct Local
      {
         ... Трамплинная функция остается прежней ...
         static void TrampolineR(BaseRhs& rhs, BaseLhs& lhs)
         {
            return Trampoline(lhs, rhs);
         }
      };
      Add<ConcreteLhs, ConcreteRhs>(&Local::Trampoline);
      if (symmetric)
      {
         Add<ConcreteRhs, ConcreteLhs>(&Local::TrampolineR);
      }
   }
};
```

---

[5] Функцию FnDispatcher::Add нельзя использовать только для регистрации динамически загружаемых функций. Для того чтобы это стало возможно, в программу нужно внести небольшие изменения.

Симметричная диспетчеризация в классе FnDispatcher реализована на уровне функций — для каждой зарегистрированной функции принимается отдельное решение.

11.8. Двойная диспетчеризация функторов

Трюк с трамплинными функциями хорошо работает с нестатическими функциями. Безымянные пространства имен предоставляют ясный способ замены статических функций нестатическими, которые невидимы за пределами данной единицы компиляции.

Однако иногда нужно, чтобы объект обратного вызова (шаблонный параметр CallbackType класса BasicDispatcher) представлял собой нечто большее, чем обычный указатель на функцию. Например, можно потребовать, чтобы каждый объект обратного вызова сохранял определенное состояние, в то время как функции могут сохранять только значения статических переменных. Следовательно, возникает необходимость регистрировать для двойной диспетчеризации *функторы*, а не функции.

Функторы (глава 5) — это классы, которые перегружают оператор вызова функции (), имитируя синтаксис вызова простой функции. Кроме того, функторы могут использовать переменные-члены для хранения своего состояния и доступа к нему. К сожалению, трюк с трамплинными функциями не применим к функторам именно потому, что функторы сохраняют свое состояние, а простые функции — нет. (А где трамплинные функции могли бы хранить свое состояние?)

Клиентский код может непосредственно использовать класс BasicDispatcher, конкретизированный соответствующим типом функтора.

```
struct HatchFunctor
{
   void operator()(Shape&, Shape&)
   {
      ...
   }
};

typedef BasicDispatcher<Shape, Shape, void, HatchFunctor>
   HatchingDispatcher;
```

Функция HatchFunctor::operator() не является виртуальной, поскольку классу BasicDispatcher нужен функтор, имеющий семантику значений, а она плохо согласуется с динамическим полиморфизмом. Однако функция HatchFunctor::operator() может переадресовать вызов виртуальной функции.

Гораздо хуже, что клиент теряет возможности автоматизации, предоставленные диспетчером, — приведение типов и поддержку симметрии.

Похоже, что мы сделали шаг назад, но только если вы не читали главу 5 об обобщенных функторах. В этой главе определен класс Functor, способный содержать любые функторы и указатели на функции, а также объекты самого класса Functor. Можно даже определять специализированные объекты класса Functor, выводя их из класса FunctorImpl. В классе Functor можно определить приведение типов. Поместив эти средства в библиотеку, можно легко реализовать симметричную диспетчеризацию.

Определим класс FunctorDispatcher для любых объектов класса Functor. Этот диспетчер содержит класс BasicDispatcher, хранящий внутри себя объекты класса Functor.

```
template <class BaseLhs, class BaseRhs = BaseLhs,
   typename ResultType = void>
class FunctorDispatcher
{
```

```
    typedef Functor<ResultType,
        TYPELIST_2(BaseLhs&, BaseRhs&)>
        FunctorType;
    typedef BasicDispatcher<BaseLhs, BaseRhs, ResultType,
        FunctorType>
        BackEndType;
    BackEndType backEnd_;
public:
    ...
};
```

Класс FunctorDispatcher использует конкретизацию класса BasicDispatcher в качестве выходного буфера. Класс BasicDispatcher хранит объекты типа FunctorType. Они представляют собой объекты класса Functor, получающие два параметра (классов BaseLhs и BaseRhs) и возвращающие объект класса ResultType.

Функция-член FunctorDispatcher::Add определяет специализированный класс Adapter, производный от класса FunctorImpl. Этот класс предназначен для приведения типов. Иными словами, он преобразовывает типы аргументов BaseLhs и BaseRhs в типы SomeLhs и SomeRhs (этот процесс называется *адаптацией*).

```
template <class BaseLhs, class BaseRhs = BaseLhs,
    ResultType = void>
class FunctorDispatcher
{
    ... как и раньше ...
    template <class SomeLhs, class SomeRhs, class Fun>
    void Add(const Fun& fun);
    {
        typedef
            FunctorImpl<ResultType, TYPELIST_2(BaseLhs&, BaseRhs&)>
            FunctorImplType;
        class Adapter : public FunctorImplType
        {
            Fun fun_;
            virtual ResultType operatop()(BaseLhs& lhs, BaseRhs& rhs)
            {
                return fun_(
                    dynamic_cast<SomeLhs&>(lhs),
                    dynamic_cast<SomeRhs&>(rhs));
            }
            virtual FunctorImplType* Clone()const
            { return new Adapter; }
        public:
            Adapter(const Fun& fun): fun_(fun) {}
        };
        backEnd_.Add<SomeLhs, SomeRhs>(
            FunctorType(FunctorImplType*)new Adapter(fun));
    }
};
```

Класс Adapter делает то же, что и трамплинная функция. Поскольку функтор имеет состояние, класс Adapter содержит объекты класса Fun — для трамплинной функции это было невозможно. Функция-член Clone, имеющая очевидную семантику, нужна для класса Functor.

Функция FunctorDispatcher::Add имеет широкое применение. Ее можно использовать для регистрации не только указателей на функции, но и почти любого функтора, даже обобщенного. Для этого нужно лишь, чтобы тип Fun в функции Add допускал применение

оператора () с аргументами типов SomeLhs и SomeRhs, а тип возвращаемого значения позволял преобразование в класс ResultType. В следующем примере показано, как регистрируются два разных функтора для объекта FunctorDispatcher.

```
typedef FunctorDispatcher<Shape> Dispatcher;
struct HatchRectanglePoly
{
   void operator(Rectangle& r, Poly& p)
   {
      ...
   }
};
struct HatchEllipseRectangle
{
   void operator(Ellipse& r, Rectangle& p)
   {
      ...
   }
};
...
Dispatcher disp;
disp.Add<Rectangle, Poly>(HatchRectanglePoly());
disp.Add<Ellipse, Rectangle>(HatchEllipseRectangle());
```

Два этих функтора никак не связаны между собой (хотя являются потомками одного и того же базового класса). Они лишь реализуют оператор () для типов, которые они должны обрабатывать.

Реализация симметрии класса FunctorDispatcher аналогична реализации симметрии класса FnDispatcher. Функция FunctorDispatcher::Add определяет новый объект ReverseAdapter, который выполняет приведение типов и меняет порядок вызовов.

11.9. Преобразование аргументов: static_cast или dynamic_cast?

Ранее для приведения типов всегда применялся безопасный оператор dynamic_cast. Однако эта безопасность достигалась за счет снижения эффективности программы.

В момент регистрации мы уже знаем, что наша функция или функтор будут активированы для пары конкретных, точно известных типов. Таким образом, при обнаружении элемента, хранящегося в ассоциативном массиве, механизму двойной диспетчеризации *известны* фактические типы. Таким образом, излишне повторять проверку типов с помощью оператора dynamic_cast, если простой оператор static_cast позволяет достичь тех же результатов за меньшее время.

Однако существуют две ситуации, в которых оператор static_cast может оказаться неприемлемым, вынуждая прибегнуть к динамическому приведению типов с помощью оператора dynamic_cast. Первая ситуация возникает при использовании виртуального наследования. Рассмотрим следующую иерархию классов.

```
class Shape { ... };
class Rectangle : virtual public Shape { ... };
class RoundedShape : virtual public Shape { ... };
class RoundedRectangle : public Rectangle,
   public RoundedShape { ... }
```

На рис. 11.2 показаны отношения между классами, входящими в эту иерархию.

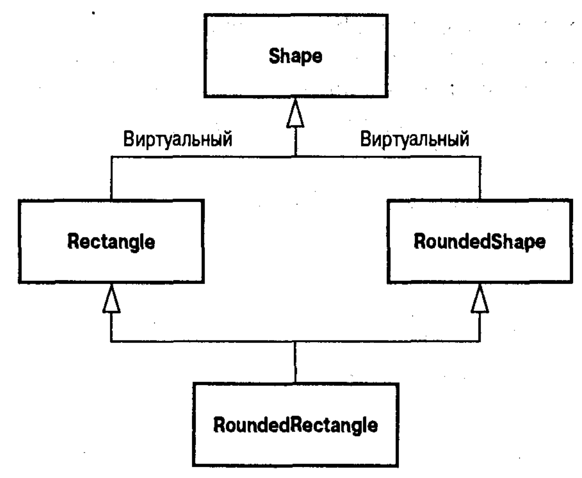


Рис. 11.2. "Бриллиантовая" иерархия классов, использующая виртуальное наследование

Возможно, это не самая изящная иерархия классов, но ведь мы никогда не знаем, что может понадобиться пользователю. Существуют вполне реальные ситуации, в которых необходимы именно "бриллиантовые" иерархии классов, несмотря на все их недостатки. Следовательно, двойной диспетчер, определенный нами, должен работать с такими иерархиями.

На самом деле диспетчер по-прежнему вполне справляется с заданием. Однако, если заменить оператор `dynamic_cast` оператором `static_cast`, при попытке привести тип `Shape&` к любому из типов `Rectangle&`, `RoundedShape&` или `RoundedRectangle&` возникнет ошибка компиляции. Это происходит потому, что виртуальное наследование отличается от простого наследования. Виртуальное наследование предоставляет средства, позволяющие нескольким производным классам совместно использовать один и тот же объект базового класса. Компилятор не может просто поместить объект производного класса в память, намертво связав его с объектом базового класса.

В некоторых реализациях множественного наследования каждый производный класс хранит указатель на свой базовый объект. Во время приведения производного типа к базовому компилятор использует этот указатель. Однако объект базового класса не хранит указатель на объекты производных классов. С прагматичной точки зрения все это значит, что после приведения объекта производного типа к виртуальному базовому типу оказывается, что механизма, позволяющего вернуться обратно к объекту производного типа, компилятор не имеет. Невозможно выполнить оператор `static_cast` для перехода от объекта виртуального базового типа к объекту производного типа.

Однако оператор `static_cast` использует более сложные средства для распознавания отношений между классами и прекрасно работает даже с виртуальными базовыми типами. Короче говоря, для иерархий, использующих виртуальное наследование, следует применять оператор `dynamic_cast`.

Вторая ситуация возникает, когда иерархия классов использует простое (даже не виртуальное) множественное наследование.

```
class Shape { ... };
class Rectangle : public Shape { ... };
class RoundedShape : public Shape { ... };
```

```
class RoundedRectangle : public Rectangle,
   public RoundedShape { ... }
```

На рис. 11.2 показаны отношения между классами, входящими в эту иерархию.

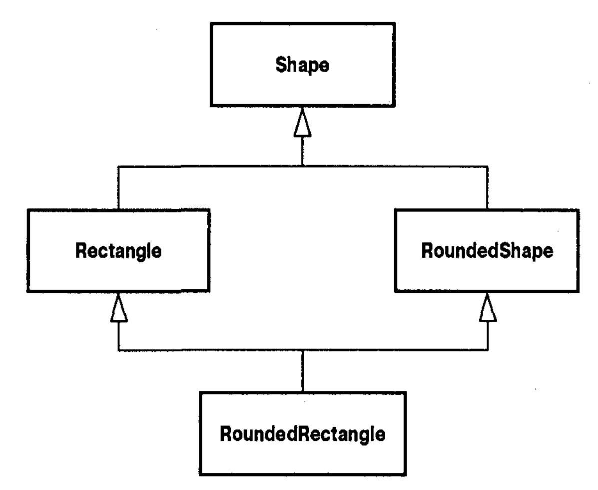


Рис. 11.3. "Бриллиантовая" иерархия классов, использующая невиртуальное наследование

Несмотря на то что форма иерархии осталась прежней, структура объектов значительно отличается от предыдущей. Объекты класса `RoundedRectangle` теперь могут быть двумя разными подобъектами типа `Shape`. Это значит, что преобразование объекта типа `RoundedRectangle` в объект типа `Shape` становится неоднозначным. Какой тип `Shape` имеется в виду — тот, который относится к типу `RoundedShape`, или тот, который связан с типом `Rectangle`? Аналогично мы не можем выполнить даже статическое приведение типа `Shape&` к типу `RoundedRectangle&`, поскольку компилятор не знает, какой из подобъектов типа `Shape` имеется в виду.

Что делать? Рассмотрим следующий код.

```
RoundedRectangle roundRect;
Rectangle& rect = roundRect;          // Однозначное неявное
                                         преобразование
Shape& shape1 = rect;
RoundedShape& roundShape = roundRect; // Однозначное неявное
                                         преобразование
Shape& shape2 = roundShape;
SomeDispatcher d;
Shape& someOtherShape = ...;
d.Go(shape1, someOtherShape);
d.Go(shape2, someOtherShape);
```

Здесь важно то, что диспетчер использует оператор `dynamic_cast` для преобразования типа `Shape&` в тип `RoundedShape&`. Если попытаться зарегистрировать трамплинную функцию для такого преобразования, возникнет ошибка компиляции.

Если диспетчер использует оператор динамического приведения типов, никаких проблем вообще не возникает. Оператор `dynamic_cast<RoundedRectangle&>` применяется к любым подобъектам базового класса `Shape`, позволяя получить правильный объект. Очевидно, что лучше динамического приведения типов ничего нет. Оператор `dynamic_cast` разработан для получения правильных объектов в иерархии классов, независимо от сложности их структуры.

Итак, оператор static_cast нельзя применять для двойной диспетчеризации иерархии классов, в которую базовый класс входит несколько раз, независимо от того, виртуально наследование или нет.

Может показаться, что оператор dynamic_cast следует применять при создании любых диспетчеров. Однако существуют два дополнительных момента.

- Очень немногие реальные иерархии классов используют "бриллиантовую" форму наследования. Такие иерархии слишком сложны, и проблемы, которые они порождают, перевешивают их достоинства. Поэтому разработчики обычно избегают применять такие иерархии классов.
- Оператор dynamic_cast выполняется намного медленнее, чем оператор static_cast. Его мощь достигается за счет снижения быстродействия. Многие клиенты работают с очень простыми иерархиями классов и хотят, чтобы их программы работали быстро. Использование оператора dynamic_cast ставит клиентов перед выбором: разработать совершенно новый диспетчер или внести изменения в библиотеку.

В библиотеке Loki реализована стратегия CastingPolicy. Она представляет собой шаблонный класс с двумя параметрами: исходный и результирующий типы. Стратегия содержит единственную статическую функцию Cast. Ниже приводится определение класса DynamicCast.

```
template <class To, class From>
struct DynamicCaster
{
   static To& Cast(From& obj)
   {
      return dynamic_cast<To&>(obj);
   }
};
```

Диспетчеры FnDispatcher и FunctorDispatcher используют стратегию CastingPolicy, следуя принципам, сформулированным в главе 1. Ниже приводится модифицированный класс FnDispatcher (изменения выделены полужирным шрифтом).

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   ResultType = void,
   template <class, class> class CastingPolicy = DynamicCaster
>
class FunctorDispatcher
{
   ...
   template <class SomeLhs, class SomeRhs, class Fun>
   void Add(const Fun& fun)
   {
      class Adapter : public FunctorType::Impl
      {
         Fun fun_;
         virtual ResultType operator()(BaseLhs& lhs,
            BaseRhs& rhs)
         {
            return fun_(
               CastingPolicy<SomeLhs, BaseLhs>::Cast(lhs),
               CastingPolicy<SomeRhs, BaseRhs>::Cast(rhs);
         }
```

```
        ... как и раньше ...
      };
      backEnd_.Add<SomeLhs, SomeRhs>(
         FunctorType(new Adapter(fun));
   }
};
```

Обратите внимание на то, что по умолчанию стратегия конкретизируется классом DynamicCast.

В заключение попробуем воспользоваться стратегией приведения типов для достижения интересных результатов. Рассмотрим иерархию классов, изображенную на рис. 11.4. Одна ее часть не использует "бриллиантовое" наследование, поэтому для нее можно смело применять оператор static_cast. Например, оператор static_cast успешно преобразует тип Shape& в тип Triangle&. С другой стороны, оператор static_cast нельзя применять для приведения типа Shape& к типу Rectangle& и любому производному от него типу, здесь нужно использовать оператор dynamic_cast.

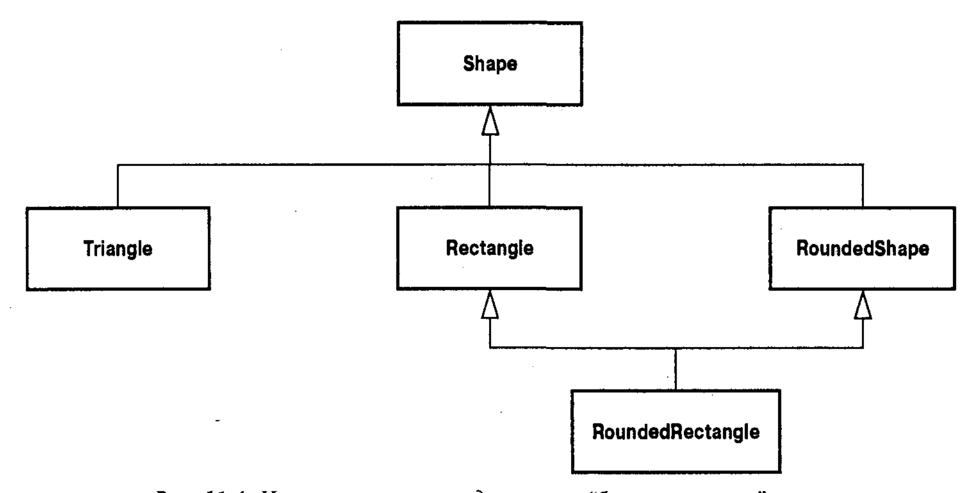


Рис. 11.4. Иерархия классов, содержащая "бриллиантовую" часть

Допустим, что мы определяем свою собственную стратегию приведения типов для данной иерархии классов. Назовем ее SuperCast. По умолчанию в ней можно применять оператор dynamic_cast, специализируя эту стратегию для отдельных ситуаций.

```
template <class To, class From>
struct ShapeCaster
{
   static To& Cast(From& obj)
   {
      return dynamic_cast<To&>(obj);
   }
};

template<>
class ShapeCaster<triangle, Shape>
{
   static Triangle& Cast(Shape& obj)
   {
      return static_cast<Triangle&>(obj);
   }
};
```

Теперь мы во всеоружии — когда можно, используется быстрое приведение типов, а в остальных случаях — безопасное.

11.10. Мультиметоды с постоянным временем выполнения

Статические диспетчеры слишком ограничены, а динамические слишком медленны. Что делать, если нам нужен быстрый и масштабируемый диспетчер?

В этом случае нужно переработать свои классы, открыв их для вмешательства двойного диспетчера. Эта возможность открывает новые перспективы. Поддержка приведения типов остается без изменения, однако средства для хранения и обнаружения обработчиков придется изменить (ведь они имеют логарифмическое время выполнения).

Чтобы разработать более совершенный механизм диспетчеризации, вновь зададимся вопросом: что такое двойная диспетчеризация? Ее можно интерпретировать как поиск функции (или функтора) на плоскости. На одной из осей отложены типы левого оператора, а на другой — правого. Найдя пересечение двух типов, мы отыщем нужную функцию. На рис. 11.5 показана двойная диспетчеризация для двух иерархий классов — `Shape` и `DrawDevice`. Обработчиками являются функции рисования, которые знают, как изобразить объект класса `Shape` с помощью объекта `DrawingDevice`.

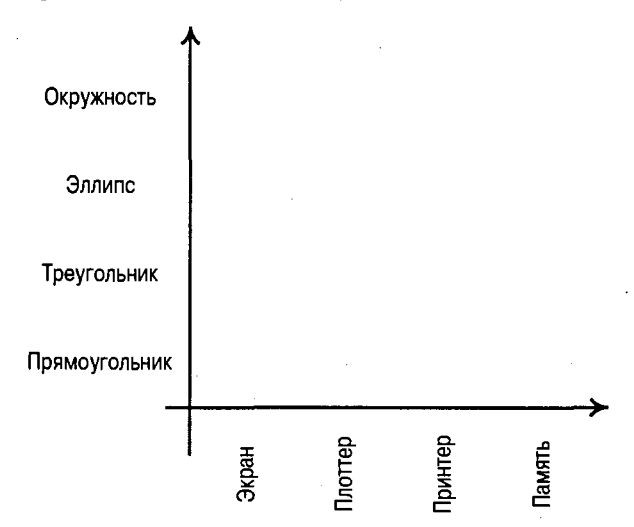


Рис. 11.5. Диспетчеризация иерархий Shape и DrawingDevice

Нетрудно догадаться, что для достижения постоянного времени поиска точки на плоскости следует использовать индексированный доступ к элементам двумерной матрицы.

Идея возникает немедленно: каждый класс должен ассоциироваться с уникальным целым числом, представляющим собой индекс в матрице диспетчера. Доступ к этому числу из любого класса должен осуществляться за постоянное время. Для этого можно применить виртуальную функцию. При двойной диспетчеризации вызова диспетчер извлекает два индекса из двух объектов, обращается к обработчику, записанному в матрице, и активирует его. Это связано со следующими затратами: два виртуальных вызова, одна операция индексирования матрицы и вызов функции через указатель на нее. На это уходит постоянное время.

На первый взгляд идея прекрасна, однако не все ее детали легко реализовать. Например, поддержка индексирования скорее всего окажется неудобной. Для каждого класса нужно задать уникальный идентификационный номер, надеясь, что компилятор сможет обнаружить возможные дубликаты. Отсчет этих номеров следует начинать с нуля и не делать пропусков, в противном случае в матрице возникнут пустые участки.

Намного лучше поручить индексирование самому диспетчеру. Каждый класс должен хранить свою статическую целочисленную переменную. Ее исходное значение должно равняться −1, что означает "не присвоенный номер". Виртуальная функция должна возвращать ссылку на это целочисленное значение, позволяя диспетчеру изменять его в ходе выполнения программы. При добавлении в матрицу нового обработчика диспетчер должен проверить его идентификационный номер. Если он равен −1, ему следует присвоить следующий доступный индекс матрицы.

Ниже приведена основная часть реализации — макрос, который следует поместить в каждый класс, входящий в иерархию.

```
#define IMPLEMENT_INDEXABLE_CLASS(SomeClass)
   static int& GetClassIndexStatic()\
   {\
      static int index = -1; \
      return index; \
   }\
   virtual int& GetClassIndex()\
   {\
      assert(typeid(*this) == typeid(SomeClass));\
      return GetClassIndexStatic();\
   }
```

Этот макрос следует поместить в открытый раздел каждого класса, для которого предусматривается множественная диспетчеризация.[6]

Шаблонный класс `BasicFastDispatcher` предоставляет те же функциональные возможности, что и прежний класс `BasicDispatcher`, используя другие механизмы хранения и извлечения данных.

```
 template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   typename CallbackType = ResultType (*)(BaseLhs&, BaseRhs&)
>
class BasicFastDispatcher
{
   typedef std:vector<CallbackType> Row;
   typedef std::vector<Row> Matrix;
   Matrix callbacks_;
   int columns_;
public:
   BasicFastDispatcher() : columns_(0) {}
   template <class SomeLhs, SomeRhs>
   void Add(CallbackType pFun)
   {
      int& idxLhs = SomeLhs::GetClassIndexStatic();
      if (idxLhs < 0)
```

[6] Именно *множественная*, а не только двойная. Это решение легко обобщить на случай множественной диспетчеризации.

```
        {
           callbacks_.push_back(Row());
           idxLhs = callbacks_.size() - 1;
        }
        else if (callbacks_.size() <= idxLhs)
        {
             callbacks_.resize(idxLhs + 1);
        }
        Row& thisRow = callbacks_[idxLhs];
        int idxRhs = SomeRhs::GetClassIndexStatic();
        if (idxRhs < 0)
        {
           thisRow.resize(++columns_);
           idxRhs = thisRow.size() - 1;
       }
       else if (thisRow.size() <= idxRhs)
       {
           thisRow.resize(idxRhs + 1);
       }
       thisRow[idxRhs] = pFun;
    }
};
```

Матрица обратных вызовов реализуется в виде вектора, состоящего из векторов типа MappedType. Функция BasicFastDispatcher::Add выполняет следующую последовательность действий.

- Извлекает идентификационный номер каждого класса, вызывая функцию GetClassIndexStatic.
- Выполняет инициализацию и настройку, если один или оба индекса не были проинициализированы. Для неинициализированных индексов функция Add расширяет матрицу, добавляя в нее дополнительный элемент.
- Вставляет обратный вызов в соответствующую позицию матрицы.

Переменная-член columns_ подсчитывает количество столбцов, добавленных к данному моменту. Строго говоря, эта переменная излишня. Вычисление максимальной длины строки в матрице приведет к тому же результату. Однако переменная columns_ добавлена для удобства.

Теперь легко реализовать функцию BasicFastDispatcher::Go. Основное отличие от предыдущей реализации заключается в том, что теперь функция Go использует виртуальную функцию GetClassIndex.

```
template <...>
class BasicFastDispatcher
{
   ... как и раньше ...
   ResultType Go(BaseLhs& lhs, BaseRhs& rhs)
   {
      int& idxLhs = lhs.GetClassIndex();
      int& idRhs = rhs.GetClassIndex();
      if (idxLhs < 0 || idxRhs < 0 ||
         idxLhs >= callbacks_.size() ||
         idxRhs >= callbacs_[idxLhs].size() ||
         callbacks_[idxLhs][idxRhs] == 0)
      {
         ... обработка ошибок ...
      }
```

```
        return callbacks_[idxLhs][idxRhs].callback_(lhs, rhs);
    }
};
```

Подведем итоги. Мы определили диспетчер, использующий матрицу. Он позволяет выполнить обратный вызов на постоянное время с помощью доступа к целочисленному индексу каждого класса. Кроме того, он выполняет автоматическую инициализацию данных (индексов, соответствующих классам). Пользователи класса `BasicFastDispatcher` должны добавить в каждый класс, использующий диспетчер, одну строку макроса `IMPLEMENT_INDEXABLE_CLASS(`*класс*`)`.

11.11. Классы BasicDispatcher и BasicFastDispatcher как стратегии

Класс `BasicFastDispatcher` (основанный на матрице) предпочтительнее класса `BasicDispatcher` (основанного на карте) с точки зрения быстродействия. Однако на основе класса `BasicDispatcher` были разработаны прекрасные классы `FnDispatcher` и `FunctorDispatcher`. Можно ли создать два новых класса — `FnFastDispatcher` и `FunctorFastDispatcher`, которые использовали бы класс `BasicFastDispatcher`?

Лучше попытаться адаптировать классы `FnDispatcher` и `FunctorDispatcher` так, чтобы они могли использовать класс `BasicDispatcher` или `BasicFastDispatcher` в зависимости от шаблонного параметра. Таким образом, диспетчер следует реализовать в виде *стратегии* для классов `FnDispatcher` и `FunctorDispatcher`, как это мы делали для приведения типов.

Задача преобразования диспетчера в стратегию облегчается, поскольку классы `BasicDispatcher` и `BasicDispatcher` имеют одинаковый интерфейс вызова. Таким образом, их легко менять местами, изменяя шаблонный аргумент.

Ниже приводится модифицированное объявление класса `FnDispatcher` (класс `FunctorDispatcher` объявляется аналогично).

```
template
<
    class BaseLhs,
    class BaseRhs = BaseLhs,
    typename ResultType = void,
    template <class, class>
        class CastingPolicy = DynamicCaster,
    template <class, class, class, class>
        class DispatcherBackend = BasicDispatcher
>
class FnDispatcher; // Аналогично классу FunctorDispatcher
```

Сами классы практически не изменяются.

Уточним требования, предъявляемые к стратегии `DispatcherBackend`. Прежде всего, очевидно, что эта стратегия должна быть шаблонным классом с четырьмя параметрами. Семантика этих параметров такова.

- Тип левого операнда.
- Тип правого операнда.
- Тип значения, возвращаемого обратным вызовом.
- Тип обратного вызова.

В табл. 11.1 класс BackendType представляет собой конкретизацию диспетчера, а объект backEnd_ — переменную этого типа. Таблица содержит функции, о которых мы не упоминали, — не беспокойтесь. Полное описание диспетчера должно содержать функции удаления обратных вызовов и "пассивного" просмотра без обращения к обратным вызовам. Они реализуются тривиальным образом. Их описание приведено в библиотеке Loki в файле Multimethods.h.

Таблица 11.1. Требования, предъявляемые к стратегии DispatcherBackend

Выражение	Тип возвращаемого значения	Примечания
copy, assign, swap, destroy		Семантика значений
backEnd.Add<SomeLhs, SomeRhs>(callback)	void	Добавляет обратный вызов в объект backEnd для типов SomeLhs и SomeRhs
backEnd.Go(BaseLhs&, BaseRhs&)	ResultType	Выполняет просмотр и диспетчеризацию для двух объектов. Если обработчик не найден, генерирует исключительную ситуацию std::runtume_error
backEnd.Remove<SomeLhs, SomeRhs>()	bool	Удаляет обратный вызов для типов SomeLhs и SomeRhs. Если обратный вызов существовал, возвращает значение true
backEnd.HandlerExists <SomeLhs, SomeRhs>()	bool	Если обратный вызов для типов SomeLhs и SomeRhs зарегистрирован, возвращает значение true. Новый обратный вызов не добавляется

11.12. Перспективы

Мы стоим на пороге новых обобщений. Результаты, полученные при разработке двойного диспетчера, можно применить для реализации действительно обобщенной множественной диспетчеризации.

На самом деле это довольно просто. Мы определили три типа диспетчеров.

- Статический диспетчер, управляемый двумя списками типов.
- Диспетчер, управляемый ассоциативным массивом, содержащим пары объектов типа std::type_info[7] в качестве ключей.
- Диспетчер, управляемый матрицей, индексированной уникальными идентификационными номерами классов.

Эти диспетчеры можно легко обобщить. Статический диспетчер, управляемый двумя списками типов, можно преобразовать в диспетчер, управляемый одним спи-

[7] Эти объекты скрыты под оболочкой класса OrderedTypeInfo, облегчающего сравнение и копирование.

ском типов, элементами которого в свою очередь являются списки типов. Это вполне реально, поскольку списки типов также являются типами. Ниже приведен оператор typedef, определяющий список типов, состоящий из трех списков типов, который можно применить для тройной диспетчеризации. Интересно, что возникающий в результате список действительно легко прочесть.

```
typedef TYPELIST_3
(
   TYPELIST_3(Shape, Rectangle, Ellipse),
   TYPELIST_3(Screen, Printer, Plotter),
   TYPELIST_3(File, Socket, Memory)
)
ListOfLists;
```

Диспетчер, управляемый ассоциативным массивом, можно преобразовать в диспетчер, управляемый вектором объектов типа std::type_info (в отличие от типа std::pair). Размер этого вектора равен количеству объектов, вовлеченных в множественную диспетчеризацию. Объявление этого класса может выглядеть следующим образом.

```
template
<
   class ListOfTypes,
   typename ResultType,
   typename CallbackType
>
class GeneralBasicDispatcher;
```

Шаблонный параметр ListOfTypes представляет собой список типов, содержащий базовые типы, вовлеченные в множественную диспетчеризацию. Например, при штриховке пересечения двух фигур можно было бы применить список TYPELIST_2(Shape, Shape).

Диспетчер, управляемый матрицей, можно обобщить с помощью многомерного массива, который можно построить на основе рекурсивного шаблонного класса. Существующая схема присвоения идентификационных номеров остается неизменной. Следует отметить, что иерархию классов, используемую для двойной диспетчеризации, можно без изменения применять и для множественной диспетчеризации.

Все эти возможные расширения требуют большого объема работы. Особенно трудная проблема, связанная с множественной диспетчеризацией, заключается в том, что в языке C++ нет универсального способа представления функций, имеющих переменное количество аргументов.

В настоящий момент библиотека Loki реализует лишь двойную диспетчеризацию. Ее обобщения читатели могут попробовать осуществить самостоятельно.

11.13. Резюме

Мультиметоды — это обобщенные виртуальные функции. Система поддержки выполнения программ на языке C++ реализует диспетчеризацию виртуальных функций по одному классу, а мультиметоды осуществляют множественную диспетчеризацию в зависимости от нескольких классов одновременно. Это позволяет реализовать виртуальные функции для наборов типов, а не для отдельного типа.

Мультиметоды лучше всего реализовывать как свойство языка. В языке C++ таких средств нет, но есть несколько способов, позволяющих использовать мультиметоды в виде библиотек.

Мультиметоды нужны в приложениях, выполняющих те или иные алгоритмы в зависимости от типов одного или нескольких объектов. Обычно они применяются для

обработки столкновений или пересечений полиморфных объектов, а также отображения объектов разными устройствами.

В этой главе мы ограничились двойной диспетчеризацией. Объект, выполняющий вызов соответствующей функции, называется двойным диспетчером. Существует несколько типов дипспетчеров.

- *Статический диспетчер.* Он полагается на информацию о статических типах (предоставленную в виде списка типов) и выполняет линейный развернутый поиск соответствующего типа. Как только требуемый тип найден, диспетчер вызывает перегруженную функцию-член для обработки объекта.
- *Диспетчер, управляемый ассоциативным массивом.* Он использует ассоциативный массив, ключами которого являются объекты класса `std::type_info`. В этом массиве хранятся обратные вызовы (либо указатели на функции, либо функторы). Алгоритм распознавания типа выполняет бинарный поиск.
- *Диспетчер, управляемый матрицей.* Это самый быстрый диспетчер, однако для его применения нужно модифицировать классы (добавить макрос в каждый класс, к которому применяется диспетчеризация). Для такой диспетчеризации нужно выполнить два виртуальных вызова, набор числовых проверок и операцию доступа к элементу матрицы.

На основе указанных выше диспетчеров можно реализовать следующие функциональные возможности.

- *Автоматизированное преобразование типов.* (Не путайте с автоматическим преобразованием типов.) Для вызова диспетчеров необходимо выполнить приведение базовых типов к производным. Для этого предназначены трамплинные функции.
- *Симметрия.* Некоторые варианты использования двойной диспетчеризации являются симметричными по своей природе. Они применяются к одинаковым типам, и порядок следования параметров для них не важен. Например, обработчику столкновений все равно — ракета столкнулась с космическим кораблем или наоборот — его реакция будет одинаковой. Реализация симметрии в библиотеке позволяет уменьшить размер клиентского кода и избежать многих ошибок.

Статический диспетчер, основанный на грубом подходе, непосредственно поддерживает эти возможности, поскольку ему доступна обширная информация о типах. Остальные диспетчеры используют разные методы и добавляют новые уровни иерархии для реализации автоматизированного преобразования типов и поддержки симметрии. Двойные диспетчеры функций отличаются от двойных диспетчеров функторов. Они более эффективны.

В табл. 11.2 приведены результаты сравнения разных диспетчеров, определенных в этой главе. Очевидно, что ни один из них не идеален. Выбор оптимального диспетчера зависит от конкретной ситуации.

Таблица 11.2. Сравнение разных реализаций двойной диспетчеризации

	Статическая диспетчеризация (класс StaticDispatcher)	Логарифмическая диспетчеризация (класс BasicDispatcher)	Диспетчеризация с постоянным временем (класс BasicFastDispatcher)
Скорость для нескольких классов	Самая высокая	Средняя	Хорошая
Скорость для многих классов	Низкая	Хорошая	Самая высокая
Зависимость между классами	Сильная	Слабая	Слабая
Переключение между существующими методами	Нет	Нет	В каждый класс добавляется макрос
Безопасность компиляции	Самая высокая	Хорошая	Хорошая
Безопасность выполнения	Самая высокая	Хорошая	Хорошая

11.14. Краткий обзор двойных диспетчеров

- В библиотеке Loki определены три вида двойных диспетчеров: классы `StaticDispatcher`, `BasicDispatcher` и `BasicFastDispatcher`.
- Объявление класса `StaticDispatcher`

```
template
<
   class Executor,
   class BaseLhs,
   class TypesLhs,
   class BaseRhs = BaseLhs,
   class TypesRhs = TypesLhs,
   typename ResultType = void
>
class StaticDispatcher;
```

Класс `BaseLhs` — это базовый тип левого операнда.

Класс `TypesLhs` представляет собой список типов, содержащий набор конкретных типов, вовлеченных в двойную диспетчеризацию в зависимости от левого операнда.

Класс `BaseRhs` — это базовый тип правого операнда.

Класс `TypesRhs` представляет собой список типов, содержащий набор конкретных типов, вовлеченных в двойную диспетчеризацию в зависимости от правого операнда.

Класс `Executor` содержит функции, вызываемые после распознавания типов. Он должен содержать перегруженную функцию-член `Fire` для каждой комбинации типов, указанных в списках типов `TypesLhs` и `TypesRhs`.

Класс `ResultType` определяет тип значений, возвращаемых перегруженными функциями `Executor::Fire`. Эти значения передаются дальше в качестве результата работы функции `StaticDispatcher::Go`.

- Класс Executor должен содержать перегруженную функцию-член OnError(BaseLhs&, BaseRhs&) для обработки ошибок. Функция Executor::OnError вызывается классом StaticDispatcher при обнаружении неизвестного типа.
- Допустим, что классы Rectangle и Ellipse являются потомками класса Shape, а классы Printer и Screen — потомки класса OutputDevice.

```
struct Painter
{
   bool Fire(Rectangle&, Printer&);
   bool Fire(Ellipse&, Printer&);
   bool Fire(Rectangle&, Screen&);
   bool Fire(Ellipse&, Screen&);
   bool OnError(Shape&, OutputDevice&);
};

typedef StaticDispatcher
<
   Painter,
   Shape,
   TYPELIST_2(Rectangle, Ellipse),
   OutputDevice,
   TYPELIST_2(Printer&, Screen),
   bool
>
Dispatcher;
```

- Класс StaticDispatcher реализует функцию-член Go, получающую параметры, имеющие типы BaseLhs&, BaseRhs& и Executor& соответственно. Вот пример ее использования (в контексте предыдущих определений).

```
Dispatcher disp;
Shape* pSh = ...;
OutputDevice* pDev = ...;
bool result = disp.Go(*pSh, *pDev);
```

- Классы BasicDispatcher и BasicFastDispatcher реализуют динамические диспетчеры, позволяющие пользователям добавлять обрабатывающие функции в ходе выполнения программы.
- Класс BasicDispatcher находит обработчик за логарифмическое время. Класс BasicFastDispatcher находит обработчик за постоянное время, однако для этого пользователь должен изменить определения всех классов, вовлеченных в диспетчеризацию.
- Оба класса имеют одинаковый интерфейс.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   typename ResultType = void,
   typename CallbackType = ResultType (*)
      (BaseLhs&, BaseRhs&)
>
class BasicDispatcher;
```

Здесь класс CallbackType является типом объекта, выполняющего диспетчеризацию. Классы BasicDispatcher и BasicFastDispatcher хранят и вызывают

объекты этого типа. Все остальные параметры имеют тот же смысл, что и в классе StaticDispatcher.

- Функции, реализуемые классами BasicDispatcher и BasicFastDispatcher, перечислены в табл. 11.1.
- Кроме указанных трех основных диспетчеров, в библиотеке Loki определены еще два диспетчера: классы FnDispatcher и FunctorDispatcher. Они используют классы BasicDispatcher и BasicFastDispatcher в качестве стратегий.
- Классы FnDispatcher и FunctorDispatcher имеют одинаковые объявления.

```
template
<
   class BaseLhs,
   class BaseRhs = BaseLhs,
   ResultType = void,
   template <class To, class From>
      class CastingPolicy = DynamicCast
   template <class, class, class, class>
      class DispatcherBackend = BasicDispatcher
>
class FnDispatcher;
```

Здесь классы BaseLhs и BaseRhs — это базовые классы двух иерархий, вовлеченных в двойную диспетчеризацию. Класс ResultType задает тип значения, возвращаемого обратными вызовами и диспетчером. Класс CastingPolicy — это шаблонный класс с двумя параметрами. Он должен реализовывать функцию-член Cast, принимающую ссылку на объект класса From и возвращающую ссылку на объект класса To. Основные реализации классов DynamicCaster и StaticCaster используют операторы dynamic_cast и statis_cast соответственно. Шаблонный класс DispatcherBackend реализует тот же интерфейс, что и классы BasicDispatcher и BasicFastDispatcher, описанный в табл. 11.1.

- И класс FnDispatcher, и класс FunctorDispatcher определяют функцию-член Add или основные типы обработчиков. Для класса FnDispatcher основным типом обработчика является класс ResultType(*)(BaseLhs&, BaseRhs&). Для класса FunctorDispatcher основным типом обработчика является класс Functor<ResultType, TYPELIST_2(BaseLhs&, BaseRhs&)>. Описание класса Functor дано в главе 5.
- Кроме того, класс FnDispatcher определяет шаблонную функцию, предназначенную для регистрации обратных вызовов.

```
void Add<SomeLhs, SomeRhs,
   ResultType (*callback)(SomeLhs&, SomeRhs&),
   bool symmetric>();
```

- Если обработчик регистрируется функцией-членом Add, описанной выше, пользователь получает преимущества автоматизированного приведения типов и симметрии.
- В классе FunctorDispatcher определяется шаблонная функция-член Add.

```
template <class SomeLhs, class SomeRhs, class F>
void Add(const F& fun);
```

- Класс F может быть одним из типов, получаемых объектом класса Functor (глава 5), включая другую конкретизацию класса Functor. Объект класса F

должен допускать выполнение оператора вызова функции с аргументами, имеющими типы BaseLhs& и BaseRhs&, и возвращать значение, тип которого может быть преобразован в тип ResultType.

- Если обработчик не найден, все механизмы диспетчеризации генерируют исключительную ситуацию, имеющую тип std::runtime_error.

Приложение

Многопоточная библиотека в стиле минимализма

В многопоточной программе одновременно существует несколько точек выполнения. Это значит, что в такой программе несколько функций могут выполняться параллельно. В многопроцессорных компьютерах разные потоки выполняются одновременно в буквальном смысле этого слова. В то же время многопоточные операционные системы в однопроцессорных машинах применяют режим *разделения времени* (time slicing) — они разделяют каждый поток на короткие интервалы времени, откладывают его и предоставляют процессорное время другому потоку. Это создает у пользователя иллюзию параллельного выполнения нескольких функций. Например, текстовый процессор может проверять грамматику, пока пользователь вводит текст.

Пользователи не желают любоваться песочными часами, пока выполняются другие потоки, поэтому программисты должны разрабатывать многопоточные программы. К сожалению, обычно их очень трудно создавать и еще труднее отлаживать. Более того, многопоточность пронизывает все приложение. Создать внешнюю библиотеку, которая надежно работала бы в многопоточном режиме, невозможно, даже если сама библиотека не использует потоки.

Отсюда следует, что нельзя игнорировать вопросы, связанные с многопоточностью. (Разумеется, на самом деле можно обойтись и без этого, однако в таком случае они станут совершенно бесполезными в многопоточной среде.) Поскольку современные приложения все интенсивнее используют многопоточное выполнение, было бы глупо игнорировать многопоточность просто из-за лени.

В приложении описываются средства и приемы, позволяющие разрабатывать машинно-независимые многопоточные объектно-ориентированные приложения на языке C++. Его нельзя рассматривать как полноценное введение в многопоточное программирование. Попытка рассмотреть все вопросы, связанные с разработкой многопоточной библиотеки, была бы бесполезной и обреченной на провал. Таким образом, мы сосредоточимся лишь на минимальных абстракциях, позволяющих создавать многопоточные компоненты.

Функциональные средства для поддержки многопоточного режима, реализованные в библиотеке Loki, скудны по сравнению с огромным количеством средств, предоставляемых современными операционными системами, поскольку они ориентированы только на обеспечение безопасности работы потоков. С другой стороны, концепции синхронизации, определенные в этом приложении, находятся на более высоком уровне, чем традиционные мьютексы (mutexes) и семафоры (semaphores) и могут оказаться полезными при разработке любого объектно-ориентированного многопоточного приложения.

П.1. Критика многопоточности

Преимущества многопоточного режима в многопроцессорных машинах очевидны. Но если компьютер имеет один процессор, многопоточность выглядит довольно глупо. Зачем замедлять работу процессора, заставляя его работать в режиме разделения времени? Очевидно, что выигрыша во времени это не дает. Чудес не бывает — у нас по-прежнему лишь один процессор, поэтому в целом многопоточный режим даже немного снизит эффективность из-за дополнительных операций обмена данными и регистрации.

Причина, по которой многопоточный режим все же применяют в однопроцессорных машинах, заключается в *эффективном использовании ресурсов*. Типичный современный компьютер имеет намного больше ресурсов, чем один процессор. Он снабжен дисководами, модемами, сетевыми картами и принтерами. Поскольку все они с физической точки зрения независимы друг от друга, их ресурсы можно использовать одновременно. Например, не существует причин, запрещающих процессору производить вычисления, пока выполняется запись информации на диск или вывод результатов на принтер. Однако это оказывается действительно невозможным, если приложение и операционная система жестко связаны только с однопоточной моделью выполнения программы. Вряд вы обрадовались, если бы ваше приложение не позволило вам ничего делать, пока не завершится загрузка данных из Интернет.

Кроме того, даже процессор в какие-то моменты времени оказывается в простое. При редактировании трехмерных изображений короткие интервалы времени между движениями мыши и щелчками кажутся процессору вечностью. Было бы хорошо, если бы программа для рисования изображений могла использовать эти моменты простоя для выполнения полезной работы, например, отслеживания луча или вычисления скрытых линий.

Основной альтернативой многопоточности является *асинхронное выполнение* (asinchronous execution). Режим асинхронного выполнения программы основан на модели обратных вызовов (callback model): при выполнении операции производится регистрация функции, которую следует вызвать после ее завершения. Основным недостатком асинхронного выполнения программы по сравнению с многопоточным режимом является избыток состояний программы. Используя асинхронную модель, нельзя проследить выполнение алгоритма шаг за шагом — можно лишь хранить состояние и позволять обратным вызовам его изменять. Поддержка такого состояния весьма проблематична и легко реализуется лишь для простейших операций.

Истинная многопоточная модель лишена таких недостатков. Каждый поток имеет неявное состояние, определяемое его точкой выполнения (оператором, который выполняется потоком в данный момент). Работу потока легко проследить, как если бы он был простой функцией. В асинхронной модели программист должен сам управлять точкой выполнения. (Основной вопрос асинхронного программирования: "Где я?".) В заключение отметим, что многопоточные программы могут реализовывать синхронную модель выполнения, что уже хорошо.

С другой стороны, потоки порождают большие проблемы, поскольку они совместно используют ресурсы, например, данные, записанные в памяти. В любой момент потоки могут прерываться другими потоками (да, именно *в любой момент*, даже посреди присваивания), поэтому операции, которые мы привыкли считать атомарными, таковыми больше не являются. Неорганизованный доступ потоков к данным также может стать причиной потери информации.

В однопоточном программировании безопасность данных при входе и выходе функции обычно гарантируется. Например, оператор присваивания (`operator=`) из класса `String` считает объект класса `String` корректным только перед началом и по-

сле завершения операции. В многопоточном программировании необходимо обеспечивать корректность объекта класса `String` даже *во время* выполнения оператора присваивания, поскольку другой поток может прервать выполнение этого оператора и применить к этому объекту другую операцию. В то время как однопоточное программирование приучает программистов рассматривать функции как атомарные операции, в многопоточном программировании необходимо самому указывать, какие операции являются атомарными. Итак, многопоточные программы порождают большие проблемы при совместном использовании данных, что следует считать их недостатком.

Большинство способов многопоточного программирования основное внимание уделяют *объектам синхронизации* (synchronization objects), позволяющим обеспечивать последовательный доступ к совместно используемым ресурсам. При выполнении атомарной операции объект синхронизации захватывается. Если другие потоки пытаются захватить тот же объект синхронизации, они фиксируются. Затем данные изменяются, и объект синхронизации освобождается. В этот момент его может захватить другой поток, получив доступ к данным. Таким образом, потоки всегда работают с корректными данными.

В следующем разделе будут описаны разные способы захвата. Их перечень не исчерпывается объектами синхронизации, хотя эти объекты позволяют решить большинство задач многопоточного программирования.

П.2. Подход, реализованный в библиотеке Loki

Для решения проблем, связанных с многопоточностью, в библиотеке Loki определена стратегия `ThreadingModel`. Эта стратегия представляет собой шаблонный класс с одним аргументом. Такой аргумент является типом языка C++, для которого реализуются функциональные возможности многопоточного программирования.

```
template <typename T>
class SomeThreadingModel
{
   ...
};
```

В следующих разделах мы последовательно раскроем содержание стратегии `ThreadingModel`. В библиотеке Loki определена однопоточная модель, которая в большинстве случаев используется по умолчанию.

П.3. Атомарные операции с целочисленными типами

Допустим, что переменная `x` имеет тип `int`. Рассмотрим оператор

```
++x;
```

Может показаться странным, что мы уделяем время анализу простого оператора инкрементации, однако именно в таких ситуациях проявляются особенности многопоточного программирования — для решения простых проблем нужно приложить много усилий.

Чтобы увеличить значение переменной `x` на единицу, процессор выполняет три операции.

1. Извлекает переменную из памяти.
2. Увеличивает значение переменной в арифметико-логическом устройстве (АЛУ) процессора. Это единственное место, где выполняются операции — в памяти данных они лишь хранятся.
3. Переменная записывается обратно в память.

Поскольку первая операция считывает, вторая — модифицирует, а третья — записывает данные, они образуют тройку, известную по названием *операция чтения-модификации-записи* (read-modify-write operation — RMW).

Предположим теперь, что оператор инкрементации выполняется в компьютере с многопроцессорной архитектурой. Для того чтобы достичь максимального эффекта во время модификации данных в процессе выполнения операции чтения-модификации-записи, процессор освобождает шину запоминающего устройства. Таким образом, второй процессор получает доступ к памяти, пока первый выполняет операцию инкрементации. Это позволяет эффективнее использовать ресурсы.

К сожалению, другой процессор может применить операцию чтения-модификации-записи к той же переменной. Например, допустим, что существуют два оператора инкрементации переменной x, которая в начальный момент времени имеет значение 0, и эти операторы выполняются двумя процессорами P1 и P2 в такой последовательности.

1. Процессор P1 захватывает шину запоминающего устройства и извлекает переменную x.
2. Процессор P1 освобождает шину запоминающего устройства.
3. Процессор P2 захватывает шину запоминающего устройства и извлекает переменную x (значение которой по-прежнему равно 0). В то же время процессор P1 увеличивает значение переменной x на единицу в своем арифметико-логическом устройстве. Результат равен 1.
4. Процессор P2 освобождает шину запоминающего устройства.
5. Процессор P1 захватывает шину запоминающего устройства и записывает значение 1 в переменную x. Одновременно процессор P2 увеличивает значение переменной x на единицу в своем арифметико-логическом устройстве. Поскольку процессор P2 извлек значение 0, результат снова равен 1.
6. Процессор P2 освобождает шину запоминающего устройства.
7. Процессор P2 захватывает шину запоминающего устройства и записывает значение 1 в переменную x.
8. Процессор P2 освобождает шину запоминающего устройства.

В итоге, хотя к переменной x были применены две операции инкрементации, ее значение станет равным 1, а не 2. Это неверный результат, причем ни один процессор (поток) не сможет распознать, что операция инкрементации была выполнена неверно. В многопоточной среде ничто не является атомарным — даже простая операция инкрементации целого числа.

Есть множество способов сделать операцию инкрементации атомарной. Наиболее эффективный из них — использовать возможности процессора. Некоторые процессоры предоставляют возможности блокировки шины запоминающего устройства — операция чтения-модификации-записи выполняется как и раньше, однако во время ее выполнения шина запоминающего устройства остается заблокированной. Таким образом, процессор P2 извлекает переменную x из памяти только после того, как процессор P1 выполнит свою операцию инкрементации.

Эта низкоуровневая функциональная возможность реализуется операционной системой с помощью функций на языке C, использующих атомарные операции инкрементации и декрементации.

Обычно операционные системы определяют атомарные операции для целочисленных типов, размер которых совпадает с шириной шины запоминающего устройства — в большинстве случаев это тип `int`. Потоковая подсистема библиотеки Loki (файл `Threads.h`) определяет тип `IntType` внутри каждой реализации стратегии `ThreadingModel`.

Элементарные функции, выполняющие атомарные операции, определенные в стратегии `ThreadingModel`, имеют следующий вид.

```
template <typename T>
class SomeThreadingModel
{
public:
   typedef int IntType; // или другой тип в зависимости от платформы
   static IntType AtomicAdd(volatile IntType& lval, IntType val);
   static IntType AtomicSubtract(volatile IntType& lval, IntType val);
... аналогичные определения для функций AtomicMultiply,
    AtomicDivide, AtomicIncrement и AtomicDecrement ...
static void AtomicAssign(volatile IntType& lval, IntType val);
static void AtomicAssign(IntType& lval, volatile IntType& val)
};
```

Эти элементарные функции получают изменяемое значение в качестве первого параметра (обратите внимание на то, что они передаются по неконстантной ссылке с помощью типа `volatile`), а второй операнд (отсутствующий в случае унарных операторов) передается в качестве второго параметра. Каждая элементарная функция возвращает копию параметра типа `volatile`. Это очень полезно, поскольку позволяет контролировать фактический результат выполнения операции. Рассмотрим ситуацию, при которой переменная типа `volatile` проверяется после выполнения операции.

```
volatile int counter;
...
SomeThreadingModel<Widget>::AtomicAdd(counter, 5);
if (counter == 10) ...
```

В этом случае код не проверяет переменную `counter` сразу после сложения, поскольку другой поток может модифицировать ее в интервале между вызовами функции `AtomicAdd` и оператором `if`. Чаще всего проверять значение переменной `counter` необходимо *сразу* после вызова функции `AtomicAdd`. Для этого достаточно написать следующий код.

```
if (AtomicAdd(counter, 5) == 10) ...
```

Наличие двух разных функций `AtomicAssign` необходимо, поскольку даже операция копирования может быть не атомарной. Например, если ширина шины компьютера равна 32 бит, а тип `long` имеет размер 64 бит, для копирования переменной типа `long` понадобится дважды обращаться к памяти.

П.4. Мьютексы

Эдгар Дийкстра (Edgar Dijkstra) доказал, что в многопоточной среде планировщик потоков операционной системы должен содержать определенные объекты синхронизации. Без них написать правильное многопоточное приложение невозможно.

Мьютексы представляют собой основные объекты синхронизации, позволяющие потокам получать доступ к совместно используемым ресурсам в организованном порядке. В этом разделе дается определение мьютекса. В остальной части библиотеки Loki мьютексы непосредственно не используются. Вместо этого в ней определяются высокоуровневые средства синхронизации, которые легко реализовать с помощью мьютексов.

Мьютекс (mutex) — это аббревиатура слов **Mut**ual **Ex**clusive (взаимоисключающий), которые определяют способ функционирования этого объекта. Мьютекс предоставляет потокам взаимоисключающий доступ к ресурсу.

Основными функциями мьютекса являются функции Acquire и Release. Каждый поток, которому необходим исключительный доступ к ресурсу (например, к совместно используемой переменной), завладевает мьютексом. Только один поток может владеть мьютексом. После того как поток завладел мьютексом, остальные потоки, вызывающие функцию Acquire, переводятся в состояние ожидания (функция Acquire ничего не возвращает). Когда поток, владеющий мьютексом, вызывает функцию Release, планировщик потоков выбирает один из ожидающих потоков и передает право владения мьютексом ему.

Итак, мьютекс выступает в роли устройства последовательного доступа: часть кода между вызовом функции mtx_.Acquire() и вызовом функции mtx_.Release() является атомарной по отношению к объекту mtx_. Все попытки завладеть объектом mtx_ откладываются, пока атомарные операции не будут завершены.

Следовательно, для каждого ресурса, совместно используемого несколькими потоками, нужно предусмотреть отдельный мьютекс. В частности, такими ресурсами могут быть объекты языка C++. Каждая неатомарная операция, выполняемая над этими ресурсами, должна начинаться с захвата мьютекса, а заканчиваться его освобождением. Обратите внимание на то, что неатомарные операции содержат неконстантные функции-члены объектов, безопасных для потока.

Например, представьте себе класс BankAccount, содержащий функции Deposit и Withdraw. Эти операции представляют собой нечто большее, чем сложение и вычитание полей класса, имеющих тип double. Они также регистрируют дополнительную информацию, относящуюся к транзакции. Если к объекту класса BankAccount обращаются несколько потоков, то эти две операции должны быть атомарными. Для этого достаточно написать следующий код.

```
class BankAccount
{
public:
   void Deposit(double amount, const char* user)
   {
      mtx_.Acquire();
      ... кладем деньги на счет...
      mtx_.Release();
   }
   void Withdraw(double amount, const char* user)
   {
      mtx_.Acquire();
      ... снимаем деньги со счета ...
      mtx_.Release();
   }
private:
   Mutex mtx_;
};
```

Возможно, вы уже догадались (если не знали этого раньше), что вызовы функций Acquire и Release должны строго чередоваться, иначе это может привести к печальным последствиям. Если вы захватите мьютекс и не освободите его, то все остальные потоки, пытающиеся овладеть им, окажутся заблокированными навсегда. В предыдущем коде при реализации функций Deposit и Withdraw следует быть очень внима-

тельным, предупреждая возможные исключительные ситуации и преждевременные выходы из функций.

Для решения этой проблемы во многих прикладных интерфейсах языка C++ определяется объект Lock, который инициализируется мьютексом. Конструктор объекта Lock вызывает функцию Acquire, а его деструктор — функцию Release. Таким образом, размещая объект Lock в стеке, можно гарантировать правильное чередование вызовов функций Acquire и Release даже в исключительных ситуациях.

Для обеспечения машинной независимости в библиотеке Loki сами мьютексы не определяются. Предполагается, что вы уже используете собственную библиотеку для поддержки многопоточного режима, в которой определены свои мьютексы. Было бы глупо дублировать их функциональные возможности. Вместо этого с помощью мьютексов в библиотеке Loki реализуется высокоуровневая семантика блокировки.

П.5. Семантика блокировки в объектно-ориентированном программировании

Объекты синхронизации связаны с совместно используемыми ресурсами. В объектно-ориентированном программировании ресурсами являются объекты. Таким образом, в объектно-ориентированной программе объект синхронизации связан с объектами приложения.

Следовательно, каждый совместно используемый объект должен содержать объект синхронизации и блокировать его в каждой модифицирующей функции-члене, как это сделано в примере с классом BankAccount. Структура, в которой каждому совместно используемому объекту соответствует свой объект синхронизации, известна под названием *блокировки на уровне объектов* (object-level locking).

Однако иногда хранить отдельный мьютекс для каждого объекта накладно. В этом случае следует применять стратегию, основанную на использовании только одного мьютекса для отдельного класса.

Рассмотрим для примера класс String. Время от времени возникает необходимость блокировать его объекты. Однако мы не хотим хранить отдельный мьютекс для каждого объекта этого класса. Это привело бы к увеличению размеров объектов, и копировать их стало бы неудобно. В этом случае для всех объектов класса String можно использовать один статический мьютекс. Как только какой-либо объект класса String выполняет операцию захвата мьютекса, она блокируется во всех остальных объектах этого класса. Эта стратегия нзывается *блокировкой на уровне класса* (class-level locking).

В библиотеке Loki определены две реализации стратегии ThreadingModel: классы ClassLevelLockable и ObjectLevelLockable. Они инкапсулируют семантику блокировки на уровне объектов и на уровне класса соответственно.

```
template <typename Host>
class ClassLevelLockable
{
public:
   class Lock
   {
   public:
      Lock();
      Lock(Host& obj);
      ...
   };
   ...
```

```
};

template <typename Host>
class ObjectLevelLockable
{
public:
   class Lock
   {
   public:
      Lock(Host& obj);
      ...
   };
   ...
};
```

С технической точки зрения объект класса Lock оставляет мьютекс захваченным. Разница между этими двумя реализациями заключается в том, что объект класса ObjectLevelLockable<T>::Lock нельзя создать, не передав ему объект класса T, поскольку в классе ObjectLevelLockable используется стратегия захвата на уровне объектов.

Объект вложенного класса Lock захватывает объект (или целый класс), связанный с мьютексом, на все время своего существования.

В приложениях реализации стратегии ThreadingModel играют роль базовых классов. Механизм наследования позволяет использовать внутренний класс Lock непосредственно.

```
class MyClass : public ClassLevelLockable <MyClass>
{
   ...
};
```

Точный вид стратегии захвата зависит от реализации стратегии ThreadingModel, выбранной в качестве базового класса. Доступные реализации приведены в табл. П.1.

Таблица П.1. Реализации стратегии ThreadingModel

Шаблонный класс	Семантика
SingleThreaded	Стратегия поддержки многопоточности не применяется совсем. Классы Lock и ReadLock представляют собой пустые макеты
ObjectLevelLockable	Семантика блокировки на уровне объектов. Каждому объекту соответствует свой мьютекс. Внутренний класс Lock захватывает мьютекс (и, неявно, связанный с ним объект)
ClassLevelLockable	Семантика блокировки на уровне класса. Каждому классу соответствует свой мьютекс. Внутренний класс Lock захватывает мьютекс (и, неявно, все объекты связанного с ним типа)

Как показано в следующем примере, синхронизированные функции-члены реализуются очень легко.

```
class BankAccount : public ObjectLevelLockable<BankAccount>
{
public:
   void Deposit(double amount, const char* user)
   {
      Lock lock(*this);
      ... кладем деньги на счет...
      mtx_.Release();
   }
```

```
    void Withdraw(double amount, const char* user)
    {
       Lock lock(*this);
       ... снимаем деньги со счета ...
    }
    ...
};
```

Проблем, связанных с исключительными ситуациями и преждевременным выходом из функций, больше не существует. Правильное чередование операций захвата и освобождения гарантируется инвариантами языка.

Универсальный интерфейс, обеспечиваемый фиктивным интерфейсом `SingleInheritance`, гарантирует синтаксическую корректность. Можно написать код, использующий многопоточный режим, а затем изменять проектные решения, просто меняя потоковую модель.

Стратегия `ThreadingModel` используется в главе 4, *Размещение в памяти небольших объектов*, главе 5, *Обобщенные функторы*, и главе 6, *Реализация класса Singleton*.

П.6. Модификатор volatile

В языке С++ предусмотрен модификатор типа `volatile`, с помощью которого любую переменную можно предоставить в распоряжение нескольких потоков. Однако в однопоточной модели этот модификатор лучше не использовать, поскольку он не позволяет компилятору выполнять некоторые важные виды оптимизации.

По этой причине в классе Loki определен внутренний класс `VolatileType`. Внутри класса `SomeThreadingModel<Widget>` класс `VolatileType` вычисляет объект типа `volatile Widget` для классов `ClassLevelLockable` и `ObjectLevelLockable`, а также простой объект класса `Widget` для класса `SingleThreaded`. Пример его использования был описан в главе 6.

П.7. Семафоры, события и другие полезные вещи

На этом мы завершаем обсуждение средств поддержки многопоточности в библиотеке Loki. Обычные многопоточные библиотеки предоставляют программистам намного более разнообразные наборы объектов синхронизации и функций, например, семафоры, события и барьеры памяти (memory barriers). Кроме того, в библиотеке Loki нет функций, активирующих новый поток. Это еще раз подтверждает тот факт, что библиотека Loki обеспечивает безопасное выполнение потоков, но не использует сами потоки.

Возможно, в будущих версиях этой библиотеки будет реализована полная потоковая модель. Многопоточность — это область, в которой обобщенное программирование проявляет всю свою мощь. Однако в ней сильно развита конкуренция — в качестве прекрасной машинно-независимой библиотеки для поддержки многопоточного режима можете применять среду ACE (Adaptive Communication Environment — адаптивная среда для обмена информацией) (Schmidt, 2000).

П.8. Резюме

В стандарте языка С++ потоков нет. Однако вопросы, связанные с синхронизацией многопоточных программ, значительно влияют на разработку приложений и библиотек. Разные операционные системы поддерживают разные потоковые модели. По

этой причине в библиотеке Loki определен высокоуровневый механизм синхронизации, поддерживающий минимальное взаимодействие с внешней потоковой моделью.

Стратегия `ThreadingModel` и три шаблонных класса, реализующих ее, образовывают основу для создания обобщенных компонентов, поддерживающих разные потоковые модели. На этапе компиляции можно выбирать стратегию блокировки на уровне объектов, стратегию блокировки на уровне классов или не применять блокировку вообще.

Стратегия блокировки на уровне объектов создает отдельный объект синхронизации для каждого объекта в приложении. Стратегия блокировки на уровне класса создает отдельный объект синхронизации для каждого класса в приложении. Первая стратегия обладает более высоким быстродействием, вторая использует меньше ресурсов.

Библиография

Alexandrescu, Andrei. 2000a. Traits: The else-if-then of types. C++ *Report*, April.

Alexandrescu, Andrei. 2000b. On mapping between types and values. C/C++ *Users Journal*, October.

Austern, Matt. 2000. The standard librarian. C++ *Report*, April.

Ball, Steve, and John Miler Crawford. 1998. Channels for inter-applet communication. *Dr. Dobb's Journal*, September. http://www.ddj.com/articles/1998/9809/9809a/9809a.htm.

Boost. The Boost C++ Library. http://www.boost.org.

Coplien, James O, 1992. *Advanced* C++ *Programming Styles and Idioms.* Reading, MA: Addison-Wesley.

Coplien, James O, 1995. The column without a name: A curiously recurring template pattern. C++ *Report*, February.

Czarnecki, Krzysztof, and Ulrich Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Reading, MA: Addisom-Wesley.

Gamma, Erich, Richard Helm, Ralh Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley. (Русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб., "Питер". — 2001. — 368 с.)

Järvi, Jaakko. 1999a. *Tuples and Multiple Return Values in* C++. TUCS Technical Report No. 249, March.

Järvi, Jaakko. 1999b. The Lambda Library. http://lambda.cs.uti.fi.

Knuth, Donald E. 1998. *The Art of Computer Programming.* Vol. 1. Reading, MA: Addison-Wesley. (Русский перевод: Кнут Дональд Эрвин. Искусство программирования. Т. 1, Основные алгоритмы, 3-е изд. — М.: Издательский дом "Вильямс", 2000. — 720 с.)

Lippman, Stanley B. 1994. *Inside the* C++ *Object Model.* Reading, MA: Addison-Wesley.

Martin, Robert. 1996. *Acyclic Visitor.* http://objectmentor.com/publications/acd.pdf

Meyers, Scott. 1996a. *More Effective* C++. Reading, MA: Addison-Wesley. (Русский перевод: Мейерс С. Наиболее эффективное использование C++. — М.: ДМК, 2000. — 304 с.)

Meyers, Scott. 1996b. Refinements to smart pointers. C++ *Report*, November-December.

Meyers, Scott. 1996c. Effective C++, 2nd ed. Reading, MA: Addison-Wesley. (Русский перевод: Мейерс С. Эффективное использование C++. — М.: ДМК, 2000. — 240 с.)

Meyers, Scott. 1996d. auto_ptr update.http://www.awl.com/cseng/titles/0-201-63371-X /auto_ptr.html. (Примечание: прием Колвина–Гиббонса нигде не задокументирован. Заметки Мейерса представляют собой наиболее аккуратное описание решения, найденного Грегом Колвином и Биллом Гиббонсом. В нем используется класс auto_ptr, позволяющий решить проблему возвращаемого функцией значения.)

Schmidt, D. 1996. Reality check. C++ *Report*, March. http://www.cs.wustl.edu/~schmidt/ editorial-3.html.

Schmidt, D. 2000. The Adaptive Communication Environment (ACE). http://www.cs.wustl.edu/ ~schmidt/ACE.html.

Stevens, AL. 1998. Undo/Redo redux. *Dr. Dobb's Journal*, November.

Stroustrup, Bjarne. 1997. *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley (Русский перевод: Страуструп Б. Язык программирования C++. — СПб: Невский проспект-Бином, 1999. — 991 с.)

Sutter, Herb, 2000. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Reading, MA: Addison-Wesley.

Van Horn, Kevin S., 1997. Compile-time assertions in C++. *C/C++ Users Journal*, October. http://www.xmission.com/~ksvhsoft/ctassert/ctassert.html.

Veldhuizen, Todd. 1995. Template metaprograms. C++ *Report*, May. http://extreme.indiana.edu/ ~tveldhui/papers/Template-Metaprograms/meta-art.html.

Vlissides, John. 1996. To kill a singleton. C++ *Report*, June. http://www.stat.cmu.edu/~lamj/ sigs/c++-report/cpp9606.c.vlissides.html.

Vlissides, John. 1998. *Pattern Hatching*, Reading, MA: Addison-Wesley.

Vlissides, John. 1999. *Visitor in frameworks*. C++ Report, November-December.

Предметный указатель

А

Б

Д

И

К

M

Н

О

П

Р

С

У

Ф

Х

Ц

Ш